Numéro d'ordre: 1922

USTL

Année : **1997**

LIFL *Laboratoire d'Informatique Fondamentale de Lille*

THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Raymond NAMYST

PM^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières

le 13 Janvier 1997, devant la commission d'examen :

Président:	J-L. DEKEYSER	LIFL
Rapporteurs:	L. BOUGÉ	LIP
	J. BRIAT	LMC
	B. PLATEAU	LMC
Examinateurs:	C. ROUCAIROL	PR*i*SM
	J-M. GEIB	LIFL
	J-F. MÉHAUT	LIFL

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 03.20.43.47.24 Fax. 03.20.43.65.66

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

M. MIGEON Michel	EUDIL
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
M. CATTEAU Jean Pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
M. GOURIEROUX Christian	Probabilités et statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie organique
M. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

À la mémoire de mon père

Remerciements

Pour commencer, je voudrais remercier les membres du jury.

- Monsieur Jean-Luc DEKEYSER, professeur à l'université de Lille I, pour m'avoir fait l'honneur de présider ce jury.

- Monsieur Luc BOUGÉ, professeur à l'école normale supérieure de Lyon, d'avoir accepté de rapporter cette thèse. Ses nombreuses remarques constructives m'ont permis d'améliorer non seulement la qualité du document mais aussi celle de l'environnement PM^2 lui-même.

- Madame Brigitte PLATEAU, professeur au LMC de Grenoble, et monsieur Jacques BRIAT, maître de conférence au LMC de Grenoble, d'avoir bien voulu corapporter cette thèse. Leurs remarques m'ont aidé à renforcer l'argumentaire développé dans ce travail.

- Madame Catherine ROUCAIROL, professeur à l'université de Versailles, d'avoir accepté d'examiner ce travail.

- Monsieur Jean-Marc GEIB, professeur à l'université de Lille I, d'avoir accepté de diriger mes recherches. Les nombreuses discussions que nous avons eues ont permis de fixer le cadre conceptuel de ce travail. Son apport a été déterminant dans le déroulement de cette thèse.

- Monsieur Jean-François MÉHAUT, maître de conférence à l'université de Lille I et fervent supporter du Racing Club de Lens, qui m'a encadré depuis mon arrivée au laboratoire. Sa grande compétence et son suivi sans faille ont été les moteurs de ce travail, et ce tant sur le plan de la conception que sur le plan de la réalisation. Son amitié et sa confiance m'ont permis de mener à bien cette thèse dans les meilleures conditions. Je lui dois beaucoup.

Je tiens également à remercier mes collègues de travail, les membres de l'équipe Yves DENNEULIN, Cédric DUMOULIN, Christophe GRANSART, Chrystel DEGRANDE, Zouhir HAFIDI, Philippe MERLE et Jean-François ROOS, les « collègues distants » Luc COURTRAI et Pierre-André WACRENIER et tous ceux qui, par leur sympathie et leur amitié, ont fait ou font du laboratoire de Lille un lieu où il fait bon travailler.

L'environnement PM^2 ne serait pas ce qu'il est sans ses utilisateurs, et je tiens à remercier tout particulièrement Bertrand LE CUN, Nathalie FURMENTO, Nordine MELAB et Christian PEREZ pour l'intérêt qu'ils ont témoigné pour notre travail. Les nombreux échanges que nous avons eu m'ont beaucoup apporté. Non, je n'ai pas oublié Yves DENNEULIN, béta-testeur officiel de PM^2, qui a resisté aux agressions des différents bugs de la plate-forme sans jamais renoncer et dont la compétence technique m'a été d'une très grande utilité durant ce travail.

Je n'aurais certainement pas eu l'audace de me lancer dans cette aventure sans les encouragements et la persuasion de Michel LATTEUX qui a su à l'époque (et avec quelques complices) me donner une confiance dont j'avais grandement besoin. Merci Michel.

J'adresse également un merci à tous les re-lecteurs du document, qui se sont livrés à une chasse aux fautes qui n'était pas de tout repos. En particulier, je dois une fière chandelle à Nathalie REVOL, bénévole aux « relecteurs du cœur », qui a accompli un travail de correction tout simplement con-si-dé-rable.

Enfin, c'est à ma famille que j'adresse mes remerciements les plus chaleureux, pour m'avoir supporté et encouragé durant toutes ces années. En particulier, je remercie tendrement Virginie, qui a eu la patience de me supporter durant ces longs mois de rédaction, pendant lesquels elle a bien voulu assurer l'intégralité de mes « corvées de vaisselle ». Son soutien et sa compréhension m'ont aidé à surmonter les périodes les plus difficiles. Je m'arrête là, car j'ai de la vaisselle à faire...

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Si les performances des architectures parallèles et distribuées ne cessent de croître, tant sur le plan de la puissance de calcul (microprocesseurs) que sur le plan des débits de communication (réseaux), force est de constater que les applications n'en tirent que trop rarement profit. Cette caractéristique est particulièrement vraie pour les applications dites *irrégulières* [149] qui demeurent extrêmement difficiles à paralléliser efficacement avec les environnements de programmation disponibles aujourd'hui. Cet état de fait est principalement dû à la difficulté de proposer des solutions aux problèmes posés par l'« irrégularité » dans chacun des registres suivants :

Modèle de programmation La parallélisation d'une application irrégulière peut mener à la génération d'un grand nombre de tâches dont le *graphe des dépendances* est déséquilibré. Pour certaines classes d'applications, ce graphe n'est pas prévisible à l'avance.

Un environnement dédié au support de ces applications doit donc fournir un modèle de programmation favorisant l'expressivité d'un parallélisme très dynamique tout en fournissant un modèle d'exécution sous-jacent adapté aux contraintes des architectures visées.

Support exécutif La notion de processus « lourd », qui représente l'unité d'exécution de la plupart des environnements de programmation actuels, fournit une forme de parallélisme en totale inadéquation avec les exigences des applications irrégulières massivement parallèles. Cette inadéquation, due à la différence existant entre la *granularité* des traitements à effectuer (tâches de l'application) et les ressources consommées par leur prise en charge, se traduit par l'incapacité des environnements actuels à assurer une prise en charge efficace des applications irrégulières.

Un environnement dédié au support de ces applications doit donc fournir un modèle d'exécution fournissant des mécanismes de prise en charge du parallélisme présentant une *granularité* plus fine que celle des processus lourds. Autant que possible, ce mécanisme ne doit pas sacrifier les aspects relatifs à sa souplesse d'utilisation.

Équilibrage de charge L'exécution efficace sur une architecture distribuée d'un ensemble de tâches dont la durée n'est pas connue *a priori* nécessite l'utilisation de techniques d'équilibrage de charge. Lorsque le graphe de dépendance de ces tâches est également inconnu, cet équilibrage doit alors s'effectuer de manière dynamique, pendant l'exécution de l'application. En outre, l'étude comportementale des applications irrégulières

effectuée par le « groupe de travail STRATAGÈME » [149] a mis en évidence l'existence de disparités importantes entre certaines de ces applications. Cette caractéristique montre la nécessité d'*adapter* la stratégie d'équilibrage de charge en fonction de l'application visée.

Un environnement dédié au support de ces applications doit donc fournir des mécanismes basiques et « génériques » d'équilibrage dynamique de charge pouvant être contrôlés par les applications.

Ces quelques remarques mettent en évidence l'exigeant « cahier des charges » que doit s'efforcer d'atteindre un environnement de programmation parallèle prétendant supporter efficacement les applications irrégulières en contexte distribué.

1.1 Objectif de la thèse

L'objectif de cette thèse est d'apporter des éléments de réponse dans les trois domaines évoqués précédemment au travers d'une proposition d'environnement de programmation parallèle basé sur la notion de processus léger nommé PM^2 (*Parallel Multithreaded Machine*). Notre contribution se traduit par 1°) la proposition d'un modèle de conception des applications irrégulières parallèles et 2°) la réalisation d'une plate-forme d'expérimentation de ce modèle.

1.1.1 Modèle de conception

Le modèle de conception des applications proposé par l'environnement PM^2 s'articule autour de trois axes principaux :

Virtualisation La conception d'une application parallèle comportant un parallélisme massif et irrégulier ne doit pas être guidée par des caractéristiques telles que le nombre de processeurs disponibles sur l'architecture cible. Au contraire, elle doit se focaliser sur l'expression du parallélisme inhérent au problème traité, en *virtualisant* la machine sous-jacente.

Le modèle de programmation PM^2 propose deux mécanismes pour le découpage parallèle des applications, l'*appel de procédure à distance léger* et le *clonage léger*, qui assurent tous deux une expression naturelle du parallélisme tout en rendant transparente la gestion des processus légers sous-jacents.

Concurrence La virtualisation décrite précédemment ne peut être satisfaisante que si toutes les tâches éligibles à un instant donné *progressent simultanément*. Cette propriété, qui est très naturelle dans un système d'exploitation classique (par exemple dans UNIX), l'est tout autant dans une application susceptible de comporter des tâches interactives, ou même des tâches devant s'acquitter de travaux périodiques, tels que les agents d'information de certains régulateurs de charge.

Le modèle d'exécution PM^2 repose sur un partitionnement de l'ensemble des processus légers sur les processeurs de l'architecture, avec un ordonnancement préemptif assurant une exécution réellement concurrente au sein de chacune des parties.

Mobilité des activités En « détournant » les mécanismes d'appel de procédure à distance et de clonage léger, un régulateur de charge peut, en utilisant des informations fournies par PM^2, contrôler la distribution des traitements en effectuant un *placement des tâches* ad-hoc. Malheureusement, une telle stratégie n'est pas suffisante pour garantir un bon équilibrage de la charge si l'application présente un comportement dynamique irrégulier, car des déséquilibres peuvent apparaître lors de la terminaison de certaines tâches.

Le modèle d'exécution PM^2 permet la *mobilité*, c'est-à-dire le déplacement d'un nœud à un autre, d'une partie des tâches s'exécutant à un instant donné. Pour ce faire, l'environnement PM^2 fournit un ensemble de fonctionnalités permettant aux applications de commander la *migration* de processus légers pendant leur exécution.

1.1.2 Choix de réalisation

La difficulté de conception d'un environnement de programmation parallèle réside dans le délicat compromis qu'il faut adopter entre les *fonctionnalités* fournies, la *portabilité* de la plate-forme et l'*efficacité* des mécanismes impliqués. Notre démarche privilégie un certain équilibre entre ces trois points, en appuyant la réalisation de la plate-forme PM^2 sur une bibliothèque de communication classique (PVM) et sur une bibliothèque de processus légers portable conçue et développée dans le cadre de cette thèse (MARCEL) :

fonctionnalités La richesse des fonctionnalités de base proposées au travers du modèle de programmation PM^2 (migration, clonage, etc.) repose presque entièrement sur les mécanismes évolués fournis par la bibliothèque MARCEL. La plupart de ces mécanismes ne sont d'ailleurs pas disponibles dans les autres bibliothèques de processus.

portabilité Compte tenu de la disponibilité de la bibliothèque PVM sur un grand nombre d'architectures et du degré de portabilité élevé de la bibliothèque MARCEL, la plate-forme PM^2 est actuellement disponible sur six architectures et ne nécessite que des modifications mineures lors d'un portage sur une nouvelle architecture (disposant de PVM).

efficacité Si le choix de la bibliothèque de communication à la base de PM^2 obéit plutôt à une démarche favorisant les fonctionnalités et la portabilité par rapport à l'efficacité, la gestion des processus légers en revanche a été particulièrement étudiée de façon à fournir une solution sans compromis dans son cadre précis d'utilisation. Ainsi, les performances de PM^2 en terme de gestion de processus sont de tout premier ordre et le surcoût des primitives liées aux communications (appels de procédure à distance, etc.) par rapport aux communications natives reste faible.

1.2 Organisation du document

Le premier chapitre « **Cadre du travail et motivations** » présente avant toute chose le contexte de ce travail, à savoir le projet ESPACE, dont l'objectif concerne la conception et la réalisation d'un environnement complet de développement et d'exécution pour les applications irrégulières sur architectures distribuées. Après avoir décrit la structure de ce projet, nous introduisons la notion d'application irrégulière en présentant l'opération inter-PRC STRATAGÈME à laquelle nous avons participé et dont l'objectif est de proposer un cadre méthodologique pour la programmation parallèle des problèmes non-structurés. Cette présentation

nous donne l'occasion d'énumérer un certain nombre d'applications irrégulières traitées au sein de ce projet et de relater les collaborations que nous avons pu mener avec d'autres équipes de recherche françaises, à la fois dans les domaines applicatifs et exécutifs. Nous terminons ce chapitre en expliquant pourquoi les environnements de programmation classiques, fondés sur un parallélisme à gros grain, ne conviennent pas pour le support de ces applications.

Le second chapitre « **Le parallélisme à grain fin** » constitue une introduction au concept de processus léger (*thread*). Nous y présentons d'abord les principales caractéristiques d'un processus léger ainsi que les différentes stratégies d'ordonnancement possibles, en insistant sur les avantages et inconvénients de chacune des approches présentées. Nous examinons ensuite quelques contraintes rendant leur utilisation directe souvent délicate. Toujours dans le domaine applicatif, nous mentionnons les utilisations « classiques » des processus légers en expliquant à chaque fois les avantages obtenus. Enfin, nous concluons ce chapitre en exhibant les bonnes propriétés du parallélisme à grain fin pour le support des applications massivement parallèles.

Le troisième chapitre « **Parallélisme à grain fin et distribution** » est un état de l'art sur les environnements de programmation distribuée basés sur les processus légers. Nous introduisons tout d'abord les différentes possibilités d'aborder l'intégration du parallélisme à grain fin en milieu distribué. Ensuite, nous dressons le portrait des principaux environnements dont la conception s'inscrit dans cette démarche. Pour chacun de ces environnements, nous présentons ses objectifs généraux, le modèle de programmation qu'il propose ainsi que la structure de son implantation. Ce chapitre est ponctué par une synthèse à l'issue de laquelle nous mettons en évidence les lacunes des environnements actuels quant au support des applications parallèles irrégulières au sens large.

Le quatrième chapitre « **PM^2 : modèle de programmation** » décrit notre proposition en termes de modèle de programmation et de modèle d'exécution pour un environnement de programmation parallèle à base de processus légers. Nous commençons par décrire l'objectif (ambitieux) du projet ESPACE proposant de virtualiser complètement l'architecture pour le programmeur et situons l'environnement PM^2 comme une étape significative vers cet objectif. Nous détaillons ensuite, dans la logique des trois axes *virtualisation-concurrence-mobilité*, le modèle de programmation de PM^2 basé sur les mécanismes d'*appel de procédure à distance léger* et de *clonage léger* ainsi que son modèle d'exécution basé sur un *ordonnancement préemptif* des processus couplé à la possibilité de *migrer* les processus d'une machine à une autre. Nous montrons l'intérêt de chacun de ces mécanismes pour le support des applications irrégulières.

Le cinquième chapitre « **PM^2 : réalisation et performances** » donne les grandes lignes de l'implantation actuelle de notre environnement et souligne les points originaux de notre contribution dans ce domaine. L'implantation de PM^2 repose sur la bibliothèque de communication PVM et surtout sur la bibliothèque de processus légers « MARCEL » conçue et développée dans le cadre de ce travail. Après un succinct descriptif de PVM, nous abordons donc l'étude des principales caractéristiques de MARCEL dont certaines sont indispensables au fonctionnement de PM^2, puis l'implantation de PM^2 proprement dite. L'efficacité de la plupart des fonctionnalités présentées est illustrée par des mesures de performances. Nous concluons ce chapitre en montrant l'intéressant compromis *portabilité/efficacité/fonctionnalités* exhibé par l'implantation actuelle, vérifiant la faisabilité du modèle de programmation de PM^2.

Le dernier chapitre « **PM^2 : quelques applications** » présente les applications les plus significatives ayant été développées avec l'environnement PM^2, principalement par d'autres équipes de recherche. Ces travaux ont été les vecteurs de collaborations inter-équipes très

fructueuses qui ont fortement contribué à l'évolution de l'environnement PM^2. Les résultats de ces travaux sont très positifs et montrent l'apport de l'environnement PM^2 tant sur le plan de la facilité de conception que sur le plan de l'efficacité d'exécution. Ils constituent une première validation de notre approche.

Chapitre 2

Cadre du travail et motivations

Depuis plusieurs années, notre équipe de recherche s'intéresse à la définition de modèles de programmation et d'exécution basés sur le concept de processus léger (*thread*). En effet, le projet PVC [78], lancé en 1990 par Eric Delattre et Jean-Marc Geib, proposait une approche basée sur les processus légers *communicants* pour le support de langages à objets parallèles (objets actifs, acteurs). L'utilisation de l'environnement PVC comme cible pour le compilateur du langage parallèle à objets BOX [82] a montré sa bonne adéquation à ce type d'applications.

L'objectif principal du projet ayant été atteint, notre équipe s'est alors intéressée à évaluer la pertinence de l'environnement PVC pour le support d'applications parallèles comportant un degré de parallélisme important dont le comportement à l'exécution est très dynamique et non-prévisible. Les travaux entrepris en ce sens nous ont permis de constater que l'approche « *processus communicants* » n'était pas toujours adaptée, à la fois pour des raisons de modèle de programmation et pour des raisons de performances, au support de ce type d'applications.

C'est pourquoi l'étude, la conception et la réalisation d'un nouvel environnement d'exécution ont été décidées. C'est ce qui constitue actuellement le cœur du projet ESPACE.

2.1 Le projet ESPACE

Le projet ESPACE [1] a débuté en 1993 au Laboratoire d'Informatique Fondamentale de Lille (LIFL) sous la direction de Jean-Marc Geib et de Jean-François Méhaut. Il s'inscrit dans l'axe Informatique Parallèle et Distribuée (*ParDis*) du LIFL.

L'objectif global de ce projet est de définir un cadre méthodologique ainsi qu'un environnement complet d'exécution pour la programmation d'applications parallèles sur architectures distribuées (machines multiprocesseurs sans mémoire commune, réseaux de stations de travail). Les applications parallèles particulièrement visées sont celles générant à l'exécution un parallélisme massif, asynchrone et non-prévisible. De manière schématique[2], le projet ESPACE peut être représenté par une fleur, dont le cœur est constitué de l'environnement d'exécution qu'il propose et dont les pétales symbolisent les applications qu'il privilégie (figure 2.1).

Alors que l'environnement d'exécution du projet ESPACE est exclusivement conçu et développé par notre équipe, les applications qui l'utilisent sont le fruit de nombreuses collaborations avec d'autres équipes de recherche françaises. En particulier, notre participation à

1. ESPACE est l'acronyme de « Execution Support for Parallel Applications in high performance Computing Environments ».

2. Pour ne pas dire poétique.

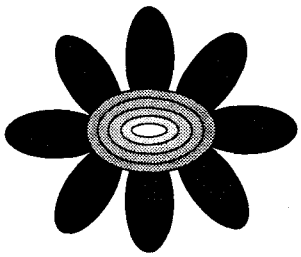

FIG. 2.1 - *Le projet* ESPACE *peut être représenté par... une fleur.*

l'opération inter-PRC STRATAGÈME, qui avait justement pour objectif de proposer un cadre méthodologique pour la programmation parallèle des problèmes non-structurés, nous a permis de travailler en étroite collaboration avec des équipes de recherche spécialisées dans certains domaines applicatifs, comme par exemple l'optimisation combinatoire ou encore les simulations discrètes. Le travail accompli durant ces collaborations a eu une grande influence sur les choix de conception effectués au niveau du support d'exécution de notre projet. Nous reviendrons plus précisément sur ces aspects dans la section 2.1.2.

2.1.1 Le cœur de la fleur

Le « cœur » du projet ESPACE, c'est-à-dire l'environnement d'exécution qu'il propose, se compose des trois concepts suivants :

- un support d'exécution dont le modèle est basé sur l'utilisation de processus légers en contexte distribué ;
- des mécanismes de régulation dynamique de la charge des applications ;
- des mécanismes d'observation (visualisation) et d'aide à l'analyse des performances des applications.

Chacun de ces concepts représente une *couche* de l'environnement ESPACE (figure 2.2).

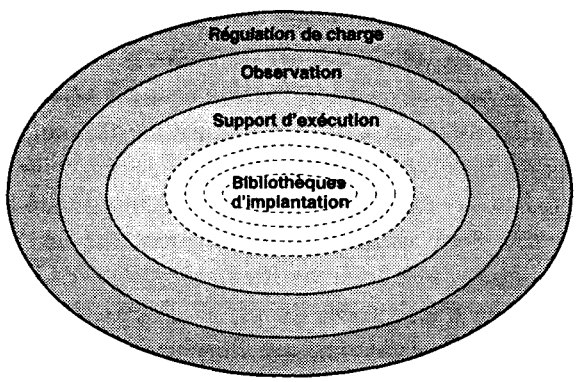

FIG. 2.2 - *Le cœur du projet* ESPACE.

2.1.1.1 Support d'exécution

Comme il a été dit en introduction, l'expérience acquise lors du projet PVC nous a permis d'entrevoir plusieurs qualités que possède le modèle des processus légers pour le support des applications parallèles en contexte distribué. La poursuite de cette étude dans le cadre du projet ESPACE a débouché sur la proposition d'un support d'exécution basé sur le multithreading distribué : PM^2 (*Parallel Multithreaded Machine*, figure 2.3).

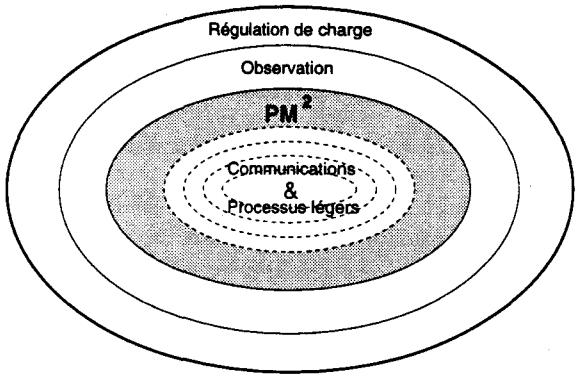

FIG. 2.3 - *PM^2 est le support d'exécution de l'environnement* ESPACE.

Les motivations, la description et l'évaluation de PM^2, qui sont l'**objet de cette thèse**, sont décrits dans les chapitres suivants. Par conséquent, nous nous contentons dans ce chapitre d'en mentionner les caractéristiques principales, sans les justifier.

D'une manière globale, les objectifs de PM^2 sont de proposer et de fournir :

- un modèle de programmation à la fois simple et puissant permettant de maîtriser le développement d'applications parallèles irrégulières distribuées en utilisant un parallélisme à grain fin ;
- une implantation portable et efficace sur de multiples architectures distribuées ;
- des outils de base facilitant au maximum la construction de régulateurs de charges spécifiques aux applications.

Modèle de programmation PM^2 fournit aux applications un modèle de programmation basé sur une extension de l'appel de procédure à distance. Chaque appel à une telle procédure donne lieu à la création d'un flot d'exécution supplémentaire. Ces flots d'exécution sont pris en charge par des *processus légers*, qui sont à la base du modèle d'exécution PM^2. Ces processus légers sont ordonnancés de manière préemptive, en fonction de leur priorité. Nous reviendrons sur toutes ces propriétés dans les chapitres suivants.

Implantation L'implantation de PM^2 s'appuie, de façon interne, sur deux bibliothèques qui regroupent les fonctionnalités dépendant de l'architecture :

- Les opérations de **communication**, nécessaires sur une architecture distribuée, sont réalisées par une bibliothèque de communication fournissant des primitives de commu-

nication point-à-point. Cette couche gère également les opérations de gestion de configuration (ajout d'une machine, retrait, etc.). Dans l'état actuel, c'est la bibliothèque PVM (Parallel Virtual Machine) [80] qui est utilisée.

- Les opérations de base relatives aux **processus légers** sont prises en charge par une bibliothèque spécifiquement conçue et développée dans notre équipe, nommée « MARCEL [3] ». Son interface constitue une extension au standard *POSIX Pthreads* et comporte en particulier des fonctionnalités telles que l'extensibilité des piles ou encore le support pour la migration de processus.

PM^2 intègre la possibilité d'exécuter des applications en contexte hétérogène (6 architectures sont pour l'instant supportées).

Support pour la répartition de charge PM^2 fournit, uniquement en contexte homogène cette fois, un mécanisme de migration de processus légers. Combiné à des fonctionnalités d'observation du système (informations de charge, liste des processus migrables, etc.), et compte tenu de son efficacité, ce mécanisme constitue un outil essentiel pour la construction de régulateurs de charge performants.

2.1.1.2 Observation

Des travaux concernant la réalisation d'outils d'observation et d'aide à l'évaluation des performances pour la plate-forme PM^2 sont en cours. Une console graphique permettant l'observation d'une application s'exécutant sous PM^2 a déjà été réalisée dans ce cadre. Celle-ci permet non seulement d'observer l'évolution des processus légers d'une application, mais aussi d'interagir avec cette dernière via une interface générique adaptable à chaque application. Cette dernière fonctionnalité est très utile lorsqu'il s'agit, par exemple, de tester le comportement d'un régulateur de charge en fonction d'un événement particulier. D'autre part, un module de prise de traces associé à des outils d'analyse sont en cours de réalisation. Cet environnement se nomme TPM^2 [4] (J.F. Roos).

2.1.1.3 Régulation de charge

De manière à rester suffisamment générale, la plate-forme PM^2 n'intègre pas de mécanismes de régulation automatique de charge. Par contre, de tels mécanismes, nécessaires au support efficace de la plupart des applications irrégulières, sont étudiés au sein du projet ESPACE. Actuellement, deux stratégies de régulation de charge sont explorées :

LBMP est un régulateur de charge spécialement adapté aux applications d'optimisation combinatoire. Sa conception s'appuie sur les fonctionnalités relatives à la migration de processus offertes par PM^2. Il constitue le travail de thèse d'Yves Denneulin [54].

MARS est un ordonnanceur de tâches pour applications parallèles ayant pour objectif l'exploitation efficace de machines par plusieurs utilisateurs sur de longues périodes de temps. Il constitue le travail de thèse de Zouhir Hafidi [87].

Ces deux outils de répartition, qui s'appuient directement sur PM^2, fournissent aux applications une interface permettant une gestion transparente des processus légers.

3. Ce qui constitue un clin d'œil à Marcel Proust pour son œuvre « *À la recherche du temps perdu* ».
4. Traced-PM^2

2.1.2 Les pétales de la fleur

Les pétales de la fleur matérialisent principalement des coopérations avec d'autres équipes de recherche utilisant PM^2 pour le support d'applications parallèles. Nous citons ici les plus significatives :

Support pour les applications d'optimisation combinatoire La bibliothèque BOB (B. Le-Cun, C. Roucairol - VERSAILLES) permet d'écrire et de tester aisément des stratégies de parallélisation, des structures de données (files de priorités) et des stratégies d'équilibrage de charge à partir d'une version séquentielle d'un algorithme *Branch & Bound* écrit avec les fonctions de cette bibliothèque. Une version de BOB utilisant PM^2 a été développée par B. Le Cun et Y. Denneulin (LILLE) [45].

Support pour le parallélisme de données Le projet « équilibrage de charge et HPF » (C. Perez, L. Bougé - LYON) vise à proposer des mécanismes automatiques de régulation de charge pour l'exécution de programmes HPF. Une des voies explorées est le multithreading comme support de cette régulation. Une évaluation très complète de PM^2 a été effectuée dans le cadre de ce projet. Cette étude a conclu que PM^2 était un bon candidat dans ce cadre [139] et les travaux actuels s'orientent vers cette solution [138].

Le projet IDOLE (B. Kokoszko, P. Marquet, J.L. Dekeyser - LILLE) développe un compilateur d'un langage data-parallèle « multi-collection » sur machine MIMD. La cible de ce compilateur est le support d'exécution PM^2 [101].

Support pour les langages fonctionnels Le projet PARALF (N. Melab, N. Devesa, M.P. Lecouffe, B. Toursel - LILLE) concerne l'évaluation parallèle de langages fonctionnels. La régulation de charge revêt une importance particulière dans ce projet. Une implantation à base de processus légers a été étudiée et réalisée en utilisant PM^2 [117, 119].

Support pour la compilation de langages à objets Le projet SLOOP (F. Baude, D. Caromel - NICE) développe une extension parallèle de C++ qui repose sur une bibliothèque de classes nommée SCHOONER. Cette bibliothèque fournit des entités proches des acteurs communiquant par messages et est développée au-dessus de PM^2 [7].

L'environnement TOSCA (C. Grenot, J.M. Geib - LILLE) est une plate-forme pour environnements coopératifs permettant de prendre en compte l'hétérogénéité des machines, ainsi que la dynamicité des connexions des applications. Cet environnement est construit au-dessus de PM^2.

Une collaboration active avec chacun de ces projets, qui nous a permis de mieux cerner les différentes exigences de ces applications, a fortement contribué à l'évolution de PM^2. Un alias de courrier électronique `pm2_users@lifl.fr` a d'ailleurs été créé pour faciliter le dialogue et les échanges d'idées entre les utilisateurs et les concepteurs de PM^2. De nouvelles collaborations sont en cours, en particulier avec le CWI d'Amsterdam sur le projet MANIFOLD (F. Arbab, P. Bouvry).

2.2 Applications parallèles irrégulières

Malgré l'apparente diversité des applications actuellement supportées par l'environnement ESPACE, elles ont toutes en commun la particularité de manipuler des structures de données

irrégulières, c'est-à-dire des structures de données (souvent très grandes) dont on ne connaît pas *a priori* l'agencement.

Par exemple, la plupart des matrices creuses constituent des structures de données irrégulières, car les valeurs significatives qu'elles contiennent ne sont pas manipulables par un schéma itératif régulier. De même, une structure arborescente générée dynamiquement telle qu'on ne puisse pas, sans l'explorer, connaître par exemple la taille de ses sous-arbres constitue une structure de données irrégulière.

Dans les sections suivantes, nous allons examiner plus précisément la notion d'application irrégulière pour comprendre les problèmes posés par la parallélisation d'une telle application. La proposition d'environnement faite dans le cadre du projet ESPACE a pour objectif d'apporter des solutions à ces problèmes.

Ces travaux se sont déroulés dans le cadre d'un projet commun à plusieurs équipes de recherches françaises : le projet STRATAGÈME, que nous présentons maintenant.

2.2.1 Le projet STRATAGÈME

Le projet STRATAGÈME avait pour objectif la proposition d'une « méthodologie de programmation parallèle pour les problèmes non-structurés ». Il a regroupé douze équipes de recherche françaises de 1993 à 1996 sous la responsabilité scientifique de Catherine Roucairol [149].

L'intérêt majeur de ce projet est d'avoir réuni des équipes de recherche travaillant dans des domaines différents, mais ayant cependant tous comme thème commun l'appréhension des problèmes irréguliers. Ainsi, des chercheurs travaillant sur des applications d'algèbre linéaire creuse, d'imagerie ou encore d'optimisation combinatoire ont pu collaborer avec des chercheurs travaillant sur la conception d'environnements d'exécution ou de régulateurs de charge.

Au-delà de simples échanges d'idées, de nouveaux concepts ont vu le jour :

Classification Les applications irrégulières ne présentent pas toutes le même « *degré d'irrégularité* » et l'on perçoit intuitivement que certaines applications ont un comportement beaucoup moins prévisible que d'autres. Une formalisation de cette constatation a été effectuée au sein de STRATAGÈME et une classification des applications irrégulières a été proposée. Nous y reviendrons dans une section ultérieure.

Outils À partir de l'investigation des différentes caractéristiques des applications étudiées au sein de STRATAGÈME, de nouveaux outils, destinés à faciliter leur implantation, ont été développés. Il s'agit principalement d'outils de placement (statique et dynamique), de bibliothèques spécialisées (BOB [45], LOREST [50]), d'outils pour la redistribution des calculs (ParList [123]) et d'environnements de programmation pour applications irrégulières (Athapascan [32], PM^2 [131]).

Expertise Grâce à la classification des applications irrégulières et à la diversité des outils proposés dans STRATAGÈME, un certain nombre d'applications réelles ont pu être « expertisées », c'est-à-dire qu'une proposition personnalisée de méthodologie de programmation a été faite pour chacune d'entre elles.

Nous allons maintenant, pour fixer les idées, citer quelques applications irrégulières concrètes, puis examiner la classification des applications selon leur « degré d'irrégularité » proposée dans le cadre de STRATAGÈME.

2.2.2 Quelques applications irrégulières

Le projet STRATAGÈME a été créé pour répondre à un besoin : celui de mieux maîtriser la programmation des applications manipulant des structures de données très irrégulières. L'enjeu est important, car ces applications sont très répandues, aussi bien dans le domaine industriel que dans le domaine de la recherche.

Les applications irrégulières que nous présentons dans cette section ont été étudiées dans le cadre de STRATAGÈME. Elles appartiennent au domaine de l'algèbre linéaire, de l'imagerie et des problèmes d'optimisation.

2.2.2.1 Algèbre linéaire

Les applications d'algèbre linéaire [63] étudiées dans le cadre du projet STRATAGÈME ont consisté en la résolution de systèmes linéaires creux de très grande taille. En particulier, un solveur à base d'*élimination de Gauss* [145], un solveur de *systèmes markoviens* [22] ainsi qu'un solveur d'*équations différentielles à coefficients rationnels* (élimination de Gauss rationnelle) ont été réalisés.

Ces algorithmes de résolution, qui manipulent des matrices de très grande taille ayant une structure creuse très irrégulière, sont très difficiles à programmer efficacement sur une architecture distribuée. Plusieurs stratégies de régulation de charge ont été proposées dans ce but. En particulier, certains travaux ont mis en évidence l'intérêt 1°) d'utiliser un prétraitement sur les matrices permettant, à moindre coût, d'obtenir une information suffisante pour partitionner les données sur les différents processeurs d'une machine ; et 2°) de supporter l'irrégularité des flots de calculs par l'utilisation de processus légers sur chaque processeur.

2.2.2.2 Imagerie

Les applications d'imagerie consistent soit en l'exploitation d'images préexistantes (*analyse d'image*), soit en la création d'images à partir d'une description numérique (*synthèse d'image*). Dans le cadre du projet STRATAGÈME, la programmation sur architecture distribuée d'une *reconstruction d'images 3D*, d'une *squelettisation 3D* [112], ou encore d'une *visualisation 3D* [29] ont été étudiées.

Comme dans les applications d'algèbre linéaire, le volume des données traitées par les applications d'imagerie est important (*i.e* > 30 Mo). Ces données, bien que ne constituant pas des structures creuses à proprement parler, présentent des régions de « densité » inégale quant à la quantité de traitement qu'elles nécessitent. Par conséquent, la parallélisation de ces applications, qui s'appuie presque toujours sur un partitionnement des données, est également délicate. Là encore, l'adoption d'une bonne stratégie d'équilibrage de charge est primordiale. La plupart des différentes politiques proposées sont basées sur une redistribution dynamique des données (par exemple l'*équilibrage élastique* [123]) en cours d'exécution. L'utilisation des processus légers est également explorée pour la prise en charge efficace de l'asynchronisme des phases de traitement [154].

2.2.2.3 Optimisation

Les applications d'optimisation étudiées dans le cadre du projet STRATAGÈME concernent la résolution parallèle de problèmes d'optimisation par des méthodes exactes ou approchées. Dans le domaine particulier de l'optimisation combinatoire, des problèmes bien connus comme

celui de l'*Affectation quadratique*, du *Voyageur de commerce* ou encore du *Sac à dos* sont très représentatifs.

Ces applications ont pour principale caractéristique d'explorer un espace de recherche [47] extrêmement vaste en vue de trouver soit un optimum (la meilleure solution au problème) dans le cas d'une méthode exacte, soit une « *bonne valeur* » dans le cas d'une méthode approchée. Contrairement aux classes d'applications précédentes, ces problèmes ne manipulent pas de larges structures de données stockées en mémoire, mais elles doivent explorer un espace de taille gigantesque (classiquement 2^{20} éléments). Il s'agit donc de partitionner cet espace de recherche de façon à pouvoir l'explorer en parallèle. Notons qu'une telle parallélisation n'a pas seulement pour objectif d'accélérer les performances de l'application, mais permet également de traiter des problèmes de taille plus élevée.

Les méthodes de résolution approchées de ces problèmes reposent sur des techniques telles que le *Recuit simulé*, la *Méthode tabou* ou encore les *Algorithmes génétiques*. Une bibliothèque générique facilitant la programmation d'applications parallèles utilisant de telles heuristiques a été développée dans le cadre de STRATAGÈME (LOREST [50]).

Les méthodes de résolution exactes reposent sur des algorithmes tels que l'*Algorithme Branch & Bound* ou l'*Algorithme A** qui manipulent des arbres (ou graphes) très irréguliers dont la taille est exponentielle par rapport à celle des données en entrée. Une bibliothèque générique permettant la parallélisation transparente des applications séquentielles de type *Branch & Bound* a également été développée dans le cadre du projet (BOB [46, 45]).

Nous reviendrons plus précisément sur le principe de la méthode *Branch & Bound* dans les sections à venir.

2.2.3 Classification

Les sections précédentes donnent un aperçu de la diversité des applications irrégulières étudiées dans STRATAGÈME. Bien que loin d'être exhaustives, elles suffisent à faire intuitivement comprendre qu'il existe plusieurs « degrés d'irrégularité » permettant de caractériser de telles applications. C'était donc un des objectifs premiers du projet STRATAGÈME que de définir de telles caractéristiques.

La proposition de classification des problèmes (ou plutôt des algorithmes) irréguliers effectuée par l'équipe de Grenoble (T. Gautier, J.L. Roch et G. Villard) dans [75] apporte un élément de réponse intéressant dans ce sens. Une définition précise de l'*irrégularité* d'un algorithme y est donnée, qui indique que « **l'irrégularité d'un algorithme est directement liée à la complexité d'ordonnancement des tâches qui le composent** ». Cette définition traduit le fait que l'irrégularité d'un algorithme est liée à la possibilité de connaître, éventuellement partiellement, le *graphe de précédence* de ses tâches.

À partir de cette définition, il est possible de distinguer trois catégories différentes d'algorithmes irréguliers [149] :

1. Le graphe de précédence est **prévisible** avant l'exécution, à partir de la seule connaissance de la taille des entrées. Dans ce cas, il est possible d'utiliser un **ordonnancement statique** de l'application pour obtenir l'équilibrage de charge désiré. C'est le cas par exemple pour certaines versions de l'algorithme de factorisation de Cholesky.

2. Le graphe de précédence est **semi-prévisible**, c'est-à-dire qu'il est prévisible par étapes, pendant l'exécution de l'application. À chaque début d'étape, il est possible de construire le graphe de précédence uniquement pour cette étape. Dans ce cas, il est possible de

faire appel, à chaque début de phase, à un ordonnanceur classique qui donnera une bonne répartition en fonction du graphe de précédence fourni. C'est le cas par exemple pour les applications d'imagerie ou de calcul formel évoquées précédemment.

3. Le graphe de précédence est **imprévisible**, c'est-à-dire que seule l'exécution du programme peut exhiber les dépendances entre les exécutions des différentes tâches. Dans ce cas il est nécessaire de faire appel à un *régulateur de charge* pendant l'exécution, pour éviter les trop grands déséquilibres. C'est le cas par exemple des algorithmes de résolution exacte des problèmes d'optimisation.

Comme on peut le constater, plus un algorithme est irrégulier, plus il nécessite l'intervention d'un régulateur de charge dynamique pour assurer son exécution efficace. Notons qu'il est parfois possible de modifier un algorithme de manière à diminuer son irrégularité (construction d'un graphe de précédence partiel, découpage en phases séquentielles), ce qui a pour conséquence de diminuer également la complexité de son ordonnancement. Bien entendu, cette diminution de l'irrégularité s'effectue au détriment des performances intrinsèques de l'algorithme et de sa facilité d'implantation, car ce dernier doit alors être modifié dans des proportions parfois importantes.

Les algorithmes irréguliers dont le graphe de précédence n'est pas prévisible constituant le domaine d'application privilégié par le projet ESPACE, l'environnement (générique) proposé vise à permettre à la fois une expression naturelle du parallélisme des applications et leur support efficace même en présence d'un comportement imprévisible.

Nous allons maintenant examiner les principaux problèmes auxquels est confrontée une telle approche.

2.2.4 Problèmes posés par l'irrégularité

Les principales difficultés posées par la parallélisation des applications irrégulières sont liées au découpage des problèmes en sous-problèmes et à la distribution des tâches résultantes sur les processeurs de l'architecture :

- Il n'est généralement pas possible de découper un problème en sous-problèmes ayant une taille équivalente (*i.e.* nécessitant un temps de traitement égal), puisqu'on ne connaît pas *a priori* le contenu de la structure de données associée. En conséquence, les découpages des problèmes irréguliers mènent à la génération de **sous-problèmes de taille variable**.

  La distribution de ces sous-problèmes sur les nœuds d'une architecture en évitant les déséquilibres de charge n'est donc pas triviale.

- La quantité de travail nécessaire au calcul d'un sous-problème n'est pas connue *a priori*, donc le découpage d'un problème en sous-problèmes peut mener à la génération de **sous-problèmes de granularité très fine.**

  La prise en charge efficace de tels sous-problèmes par un support d'exécution distribué est délicate, notamment lorsque les temps de communication ou/et de création de processus sont importants par rapport au temps de traitement de ces problèmes.

- Le nombre de sous-problèmes générés lors du découpage d'un problème n'est pas toujours contrôlable[5], ce qui peut mener à la génération d'un **degré de parallélisme**

5. ou alors de façon très artificielle

massif.

La capacité d'un environnement d'exécution à supporter le parallélisme massif en est donc une caractéristique essentielle.

Si l'on s'intéresse plus précisément aux applications irrégulières dont le graphe de précédence n'est pas prévisible, toutes les caractéristiques précédentes sont réunies, ce qui rend la présence d'un mécanisme de régulation de charge nécessaire.

Le fonctionnement et les performances d'un régulateur de charge sont très liés aux fonctionnalités fournies par le support d'exécution sous-jacent. Par exemple, un support d'exécution peut ne pas gérer efficacement l'exécution simultanée d'un nombre important de tâches parallèles, ce qui oblige le régulateur de charge à mettre en place un mécanisme de stockage des tâches en attente d'exécution pour retarder leur exécution.

De même, les fonctionnalités du support d'exécution conditionnent également la programmation des applications elles-mêmes. Par exemple, un support d'exécution ne permettant pas une exploitation efficace d'un parallélisme à grain fin oblige les applications à « grossir le grain » de parallélisme de façon artificielle.

Pour bien comprendre les contraintes posées par les applications « fortement » irrégulières, nous allons décrire un exemple caractéristique de cette classe d'applications : l'exploration d'un espace de recherche par la méthode *Branch & Bound*.

2.2.5 Exemple: méthode d'exploration de type Branch & Bound

L'algorithme *Branch & Bound* est un algorithme particulier d'exploration d'espaces de recherche typiquement utilisé pour la résolution de problèmes d'optimisation combinatoire difficiles (NP-complets). Dans ces problèmes, il s'agit de trouver, parmi un très vaste ensemble fini de solutions possibles (dont la cardinalité est exponentielle en fonction de la taille du problème), la *meilleure* solution selon un critère donné. Pour fixer les idées, on peut considérer qu'une solution est un ensemble de valeurs affectées aux variables du problème.

2.2.5.1 Algorithme

Le principe de cet algorithme consiste en la construction dynamique d'une structure arborescente mémorisant la portion de l'espace de recherche déjà explorée. Les nœuds de l'arbre de recherche ainsi construit ont la signification suivante :

- Le **nœud racine** représente l'étape initiale de l'algorithme. Aucune variable n'est encore fixée à ce stade.
- Un **nœud intermédiaire** représente une *spécialisation* de son père. Les variables fixées dans le père le sont aussi dans le fils (et de la même manière), mais de nouvelles variables (une seule la plupart du temps) sont fixées dans le fils.
- Un **nœud terminal** (une feuille de l'arbre) représente une *solution*. Toutes les variables du problème sont fixées dans un tel nœud.

Bien que la construction de cet arbre puisse s'effectuer de plusieurs manières, elle consiste toujours en la répétition des trois opérations suivantes :

sélection À chaque étape, un nœud non encore *exploré* est choisi. Selon les variantes, ce choix peut être fonction de la position du nœud dans l'arbre (stratégie *en profondeur*

d'abord ou *en largeur d'abord*) ou encore des caractéristiques de la solution partielle contenue dans le nœud (stratégie du *meilleur d'abord*). Dans ce cas, des *priorités* sont alors attachées aux nœuds (par l'opération d'évaluation) et c'est le nœud de plus grande priorité qui est choisi [58].

séparation L'exploration d'un nœud non-terminal conduit à une opération de *séparation*, c'est-à-dire à la génération d'un certain nombre de nœuds fils. Cette séparation consiste généralement au choix d'une variable non encore connue et à la génération d'autant de nœuds fils que de valeurs possibles pour cette variable.

évaluation Les nouveaux nœuds issus d'une séparation sont enfin évalués. Cette opération consiste à calculer la *valeur* de la solution partielle contenue dans un nœud. Une caractéristique fondamentale de l'algorithme de *Branch & Bound* réside dans le fait qu'à cette étape, l'exploration de certains nœuds pourra être écartée. En effet, l'évaluation d'un nœud permet généralement de fixer une *borne maximale* (en supposant qu'il s'agit d'un problème de minimisation) sous laquelle la solution optimale se trouve à coup sûr. Ainsi, lorsque l'évaluation d'un nœud produit une valeur supérieure à cette borne, on sait que le nœud ne mènera pas à la meilleure solution, ce qui permet d'élaguer l'arbre d'exploration à cet endroit (figure 2.4).

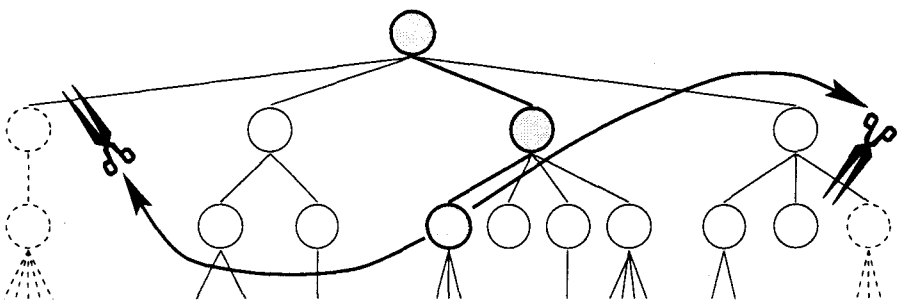

FIG. 2.4 - *Dans une exploration de type Branch & Bound, l'évaluation d'un nœud peut élaguer plusieurs branches de l'arbre.*

L'arbre d'exploration engendré par un tel algorithme est généralement très déséquilibré et son « agencement » n'est pas prévisible à l'avance. Enfin, il est important de noter que c'est de la qualité d'élagage de l'arbre que dépendent directement les performances de l'algorithme. Deux facteurs influencent cet élagage:

1. La qualité de l'évaluation de la borne supérieure. Ce facteur est intrinsèque à l'application et ne peut être calculé de façon générique par un support d'exécution.

2. La stratégie de sélection utilisée (ordre d'exploration). Ce facteur ne dépend pas de l'application et peut donc être entièrement pris en compte par une bibliothèque générique [46].

2.2.5.2 Parallélisation

La taille d'une arborescence générée par un algorithme *Branch & Bound* croît de façon exponentielle par rapport à la taille du problème. Par exemple, un problème comportant une

vingtaine de variables peut mener à la génération de plusieurs millions de nœuds et demander un temps de résolution de plusieurs centaines d'heures sur un processeur classique. Dans un tel contexte, la parallélisation de l'algorithme est intéressante car elle permet :

1. d'augmenter la puissance de calcul, ce qui permet de traiter des problèmes plus rapidement ;
2. d'augmenter la mémoire disponible, ce qui permet de traiter des problèmes encore plus grands.

Le principe de fonctionnement d'un algorithme de *Branch & Bound* parallèle consiste en l'exploration de plusieurs nœuds en parallèle. Sur une architecture multiprocesseurs à mémoire commune, cette parallélisation s'effectue aisément et nécessite seulement de protéger l'accès simultané à la file des nœuds en attente d'être explorés. Dans un contexte distribué, cette opération est beaucoup plus délicate, car il faut apporter une solution aux problèmes suivants :

- **Comment stocker la file des nœuds en attente ?** Deux solutions sont généralement mises en œuvre.

  La première consiste à utiliser une file centralisée sur un site privilégié. Lorsqu'un processeur désire sélectionner un nouveau nœud, il doit alors effectuer un accès distant à cette file virtuellement partagée, ce qui est coûteux. De même, lorsqu'un processeur a effectué une opération de séparation, l'insertion des nouveaux nœuds dans la file doit également s'effectuer par une opération distante. Cette solution souffre d'un défaut majeur : elle n'est pas généralisable à une architecture comprenant un nombre important de processeurs, en raison du goulot d'étranglement constitué par la file dans ce cas.

  La seconde solution consiste à distribuer la file sur les différents nœuds de l'architecture, de manière à accélérer les opérations d'extraction de la file (qui peuvent le plus souvent être effectuées localement). Dans ce cas, le problème consiste à garantir l'équilibrage du nombre de nœuds sur chaque site (ou à mettre en place des mécanismes d'accès distant lorsqu'un processeur n'a plus de travail localement) et à garantir, dans le cas d'une stratégie du *meilleur d'abord*, qu'un processeur (parmi n) obtienne toujours l'un des n meilleurs nœuds lors d'un retrait de la file.

- **Comment propager la valeur de la borne supérieure ?** Lorsqu'un nœud est évalué, il faut vérifier s'il est utile de l'explorer (par comparaison avec la borne supérieure trouvée jusqu'alors) et éventuellement mettre à jour la borne supérieure si ce nœud est terminal et qu'il représente le « meilleur candidat » rencontré jusqu'alors. Ces deux opérations ne sont pas triviales à réaliser sur une architecture distribuée et sont assez coûteuses à cause des communications qu'elles occasionnent. C'est pourquoi la plupart des implantations distribuées du *Branch & Bound* utilisent une *cohérence relâchée* à propos de cette variable, en n'effectuant les mises à jour globales que de manière périodique. Cette solution possède l'inconvénient de ne pas élaguer l'arbre assez souvent, mais compense cet handicap par des communications moins fréquentes. Il s'agit donc principalement de trouver un compromis entre ces deux phénomènes, c'est-à-dire une « bonne » fréquence de mise à jour de la borne supérieure.

On le voit, la parallélisation d'une application fortement irrégulière — en l'occurrence une application de *Branch & Bound* — sur une architecture distribuée est un travail complexe et délicat.

Un des objectifs de l'environnement ESPACE est de proposer un support efficace et générique facilitant la parallélisation de ce type d'applications.

À ce stade de notre étude, il est possible de dénombrer certaines propriétés qu'un environnement prétendant au support d'applications irrégulières parallèles devrait posséder. C'est ce que nous détaillons dans la section suivante.

2.3 Vers un support exécutif pour applications irrégulières

Les supports d'exécution existant pour les applications parallèles distribuées sont nombreux. Souvent, ceux-ci ne proposent que des fonctionnalités destinées à faciliter l'exploitation des architectures distribuées : des fonctionnalités de **gestion de processus** et des fonctionnalités de **communication**. Cependant, ces fonctionnalités sont insuffisantes pour supporter efficacement des applications parallèles irrégulières :

- Nous l'avons vu dans les sections précédentes, une application irrégulière peut générer des centaines de milliers de tâches potentiellement parallèles. Ce **parallélisme massif** n'est pas directement exploitable par l'architecture sous-jacente, qui se compose classiquement de quelques dizaines de processeurs. Par conséquent, à moins qu'un support d'exécution évolué ne les en décharge, c'est aux programmeurs de ces applications de réaliser cette « adaptation » qui est non seulement souvent complexe, mais s'effectue de surcroît au détriment d'une éventuelle réutilisabilité du mécanisme. En effet, le programmeur a, dans ce cas, deux possibilités : soit il grossit artificiellement le grain de parallélisme de son application, soit il met en place un mécanisme de « mise en attente » des tâches ne pouvant être exécutées dès leur génération (par exemple des *files de priorité* dans les applications d'optimisation combinatoire).

  Nous pensons qu'un bon support d'exécution pour les applications irrégulières devrait prendre en charge ces aspects de manière la plus transparente possible et offrir ainsi aux concepteurs d'applications une *machine virtuelle* capable de supporter un parallélisme massif.

- Une caractéristique somme toute très liée à la précédente est que le **parallélisme** généré par de telles applications peut être relativement **fin**, c'est-à-dire consister en un ensemble de tâches dont la durée de vie est très courte. Par conséquent, et compte tenu de leur nombre, il est primordial que les mécanismes mis en jeu par l'activation de ces tâches sur l'architecture soient très efficaces. Nous verrons dans le chapitre suivant qu'une approche classique, basée sur la création d'un nouveau processus pour chaque activation, n'est pas praticable dans ce cadre, à la fois pour des raisons d'efficacité et pour des raisons d'échelle. C'est pourquoi la plupart des applications utilisent un mode de fonctionnement basé sur un ensemble fixe de processus *serveurs de calcul* qui prennent en charge *séquentiellement* l'exécution des différentes tâches. Cependant, comme nous le découvrirons également au chapitre suivant, cette approche comporte de nombreux inconvénients qui la rendent mal adaptée au support des applications irrégulières.

  Les processus légers, dont nous examinerons les caractéristiques au chapitre 3, présentent une alternative attrayante pour résoudre ces problèmes. En effet, les temps de latence causés par les primitives qui leur sont associées sont très courts, ce qui autorise leur utilisation intensive. Un des objectifs de cette thèse est de montrer que ce mécanisme, s'il est bien maîtrisé, est tout à fait adapté au support du parallélisme à grain fin.

- Sur une architecture distribuée, la **gestion des communications** revêt souvent un caractère problématique. La raison est directement liée aux performances de celle-ci, car il n'est un secret pour personne qu'un échange de données entre deux processeurs distants est beaucoup plus coûteux qu'un échange de données qui s'effectuerait via une mémoire physiquement partagée entre les processeurs. Le problème principal est rencontré dans les applications pour lesquelles il n'est pas possible de garantir que les données à destination d'un processus lui parviennent toujours avant qu'il n'en ait besoin. Les applications irrégulières distribuées rentrent bien évidemment dans ce cadre. Dans ce cas, ce dernier est *bloqué* (par le système d'exploitation) en attendant la disponibilité des données, ce qui risque d'occasionner une période d'inactivité pour le processeur sous-jacent si aucun autre processus n'est prêt à s'exécuter.

  Pour éviter ces situations, des techniques permettant de recouvrir les communications par des calculs sont utilisées. Le principe consiste à organiser l'application de façon à ce que, sur chaque processeur, des traitements puissent être effectués pendant les situations d'attente de communications. Cette technique est utilisable soit en modifiant l'algorithme de l'application lui-même, ce qui est souvent délicat et peu réutilisable, soit en augmentant le nombre d'activités concurrentes sur chaque processeur.

  Cette dernière méthode est souvent réalisée par l'utilisation de *processus légers* (cf. applications décrites dans [63, 154]) qui permettent la coexistence efficace de plusieurs flots d'exécution au sein d'un même processus. Nous y reviendrons plus en détail au chapitre 3. Cependant, cette utilisation des processus légers n'est effectuée que dans cet unique but et ne s'intègre pas dans la logique globale de conception de l'application.

  Nous pensons qu'un mécanisme de recouvrement des communications par les calculs ne doit pas nécessiter d'intervention particulière du concepteur d'application, mais au contraire être pleinement intégré dans le modèle de programmation, de manière à favoriser sa généricité à la fois au niveau applicatif (indépendance par rapport aux applications) et au niveau exécutif (indépendance par rapport à l'architecture).

- Le caractère imprévisible de certaines applications irrégulières rend le recours à un mécanisme d'**équilibrage de charge** nécessaire pour assurer une exécution efficace. La politique de régulation peut être entièrement réalisée au niveau de l'application, ce qui optimise son efficacité (l'algorithme de régulation est parfaitement adapté à l'application) mais complique beaucoup la tâche du programmeur et limite souvent la réutilisation du mécanisme dans d'autres applications. D'un autre côté, la politique de régulation peut être entièrement réalisée au niveau du support d'exécution. Dans ce cas, le mécanisme est générique et n'utilise aucune information propre aux applications pour effectuer l'équilibrage. Les environnements de programmation DTS [20], DTMS [35] ou encore TPVM [67] en sont des exemples représentatifs. Si cette alternative a le mérite de permettre au programmeur de se focaliser sur l'expression du parallélisme de son application, elle risque en revanche de ne pas être optimale, voire d'être contre-performante dans certains cas.

  Un bon compromis entre ces deux approches est d'opter pour une régulation *intégrée* à l'environnement d'exécution [6] mais *guidée* par les applications. C'est l'approche adoptée par le projet APACHE [140] dans lequel la couche *Athapascan1* [17] fournit un mécanisme

6. classiquement sous forme d'une couche logicielle implantée au-dessus du support d'exécution

de régulation de charge guidé par le graphe de précédence des applications, et celle du projet ESPACE dans lequel la couche LBMP [54] fournit une régulation guidée par des priorités fixées par le programmeur. En fait, cette approche hérite des avantages des deux précédentes, car elle permet de déléguer les aspects « mécaniques » de la régulation au support d'exécution tout en laissant au programmeur la possibilité d'exprimer un contrôle sur celle-ci.

Cependant, quelle que soit l'approche adoptée, l'efficacité des mécanismes de régulation dépend des fonctionnalités fournies par le support d'exécution. En particulier, en l'absence d'outils permettant d'effectuer la *migration de processus* [61], un régulateur de charge ne peut qu'effectuer un *placement* des processus lors de leur création. Lorsqu'une application est réellement dynamique et imprévisible, cela n'est pas suffisant car des déséquilibres peuvent se produire sans qu'il n'y ait de création pour les compenser.

Pour toutes ces raisons, nous pensons qu'une bonne approche pour la régulation des applications irrégulières réside dans un environnement intégrant:

1. un support d'exécution fournissant des fonctionnalités suffisamment puissantes (migration, informations de charge) pour autoriser la réalisation d'un vaste éventail de régulateurs de charge;
2. un ensemble de « modules » de régulation de charge, pouvant être spécialisés pour certains types d'applications (LBMP [54], MARS [87]), permettant la conception d'applications parallèles de manière transparente à la distribution.

On le voit, la conception d'un environnement destiné au support des applications irrégulières soulève de nombreux problèmes. La plupart des environnements existants ne résolvent qu'un sous-ensemble de ces problèmes et reportent les problèmes restant à la phase de conception des applications elles-mêmes. Cette approche n'est évidemment pas satisfaisante.

Nous pensons, au contraire, que tous ces problèmes doivent être pris en charge au niveau d'un environnement d'exécution. Pour cela, il est nécessaire de définir un modèle de programmation permettant à la fois au programmeur d'application d'exprimer directement le parallélisme naturel de son application et au support d'exécution d'avoir un degré de liberté lui permettant d'exécuter cette application efficacement. L'idée principale que nous mettons en avant est de *virtualiser* au maximum l'architecture de la machine, en fournissant aux applications une machine (virtuelle) composée de plusieurs centaines de milliers de processeurs.

Dans la section suivante, nous allons passer en revue quelques environnements parallèles « à gros grain » et voir pourquoi leur utilisation ne constitue pas une solution satisfaisante pour le support des applications irrégulières, compte tenu de ce que nous venons de mettre en évidence.

2.3.1 Inconvénients du parallélisme à gros grain

À l'heure actuelle, les environnements de programmation parallèle *distribuée* les plus utilisés sont des environnements dits « à gros grain » de parallélisme, autrement dit des environnements dont l'entité d'exécution de base est le *processus lourd* (UNIX). Les environnements tels que PVM [79], MPI [121], LANDA [124] ou encore LINDA [24, 122] en sont des exemples représentatifs.

2.3.1.1 Généralités

Au-dessus de tels environnements, une application parallèle est constituée d'un ensemble distribué de processus lourds interagissant potentiellement[7] au travers d'un réseau de communication. Les interactions des processus peuvent être contrôlées de différentes manières, suivant le modèle de programmation proposé par l'environnement. Parmi les nombreux types d'interaction possibles (exécution dirigée par les données, paradigme client/serveur, etc.), les deux plus répandus dans les environnements actuels sont les suivants :

Envoi de messages L'approche « *communication par envoi de messages* » est sans aucun doute la plus utilisée sur ce type d'architecture, puisqu'elle préserve une certaine analogie avec les caractéristiques de la machine (l'échange d'informations entre processeurs ne peut s'effectuer que par canaux de communication). Différentes variantes à ce mécanisme existent et se distinguent principalement par le mode de synchronisation instauré entre les processus impliqués dans une opération de communication. Parmi ces variantes (communications synchrones [97], messages actifs [161], etc.), l'interaction par envoi *asynchrone* de messages est la plus répandue et permet aux processus de déposer des messages dans des boîtes aux lettres (ou notion dérivée) sans synchronisation particulière avec le processus destinataire. Les environnements PVM [80], MPI [121] et P4 [21] — pour ne citer que les principaux — proposent tous trois ce modèle d'interaction.

Mémoire partagée D'autres environnements proposent un mode d'interaction basé sur le partage de mémoire entre les processus. Étant donné la nature distribuée de l'architecture sous-jacente, ce partage est évidemment *virtuel* et des mécanismes destinés à fournir aux processus une « vision locale » de ces données sont utilisés de manière plus ou moins transparente. Le partage de mémoire peut s'effectuer sur la base de *pages mémoires* [110], ce qui offre aux processus une transparence maximale mais occasionne souvent des coûts opératoires importants, sur la base d'*objets* (DREAM [60], DOSMOS [18]), ce qui permet de diminuer les mises à jour en regroupant les accès mémoire au sein des méthodes d'objets, ou encore sur la base d'une structure de donnée globale particulière présentant une interface propre (LINDA [24]). Plutôt que d'utiliser une stratégie de centralisation des données dans le système qui serait fortement pénalisante du point de vue des performances, les différentes implantations utilisent typiquement des techniques de réplication des données sur différents sites, permettant des accès réellement locaux dans certains cas. Des mécanismes chargés d'assurer une certaine sémantique de *cohérence* (relâchée ou forte) des données partagées sont alors utilisés, de manière plus ou moins transparente.

Aussi différentes qu'elles puissent être d'un point de vue conceptuel, ces approches possèdent néanmoins toutes une implantation basée sur l'*échange de messages* entre les processus. C'est pourquoi, dans la suite de cette section, nous allons appuyer notre argumentation sur la base d'un modèle d'exécution spécifiquement orienté « envoi de message », sachant qu'elle s'applique aux environnements distribués à gros grain en général.

2.3.1.2 Limitations

Dans le cadre général du support des applications parallèles sur architectures distribuées, les environnements de programmation dont l'exécutif est basé sur l'utilisation de processus

7. En réalité, c'est presque toujours le cas.

lourds atteignent très vite leurs limites pour des applications dont le parallélisme est massif et irrégulier.

En effet, le processus lourd, qui représente l'unité d'exécution de la plupart des systèmes d'exploitation actuels, est une entité consommant beaucoup de mémoire et nécessitant de la part du noyau un travail important lors de sa création, lors de ses commutations et lors de sa terminaison (d'où le terme « *lourd* »). Il découle de cette caractéristique que :

- Il n'est pas possible de créer un nombre important de processus sur un même nœud, faute de quoi des mécanismes de va-et-vient de la mémoire centrale vers la mémoire secondaire viendraient entamer sérieusement les performances de l'application. Autrement dit, les applications ne peuvent comporter qu'un faible nombre de processus à un instant donné.

- Le coût d'une création et d'une terminaison de processus est suffisamment élevé pour dissuader l'utilisation fréquente de ces fonctionnalités. Aussi, les applications utilisent généralement des configurations statiques de processus créés au cours d'une phase préliminaire à la phase de calcul proprement dite.

La conséquence de tout ceci est que la plupart des applications parallèles développées au-dessus d'environnements à gros grain s'articulent autour d'une configuration fixe d'un petit nombre de processus par nœud. Dans un contexte où ni la durée des tâches ni leur nombre[8] n'est connu à l'avance, ces processus se comportent alors comme des « serveurs de calcul » prenant séquentiellement en charge l'exécution des tâches de l'application :

```
loop
   attendre_requête;
   traiter_requête;
   retourner_résultat;
end loop;
```

Cette organisation, qui demeure la seule utilisable dans un tel contexte, présente malheureusement de nombreux inconvénients, tant sur le plan des performances que sur le plan fonctionnel :

Virtualisation Le nombre de processus utilisés dans l'application étant directement lié à la topologie de l'architecture sous-jacente, ceux-ci perdent la fonction de « *processeur virtuel* » qu'il possèdent en contexte local. De plus, ce mode de fonctionnement en « processus serveurs » favorise les schémas de type maître/esclave (pour la distribution des tâches) et diminue de ce fait l'*extensibilité* de l'application à des architectures distribuées constituées de très nombreux nœuds.

Il est possible d'éviter ce problème en distribuant complètement le contrôle de la distribution des tâches, mais au détriment de l'efficacité de l'application : soit les processus serveurs s'occupent également de la distribution de tâches, auquel cas de nombreuses communications ne seront pas « recouvertes », soit des processus spécialisés dans la distribution des tâches sont créés sur chaque nœud, auquel cas les opérations de commutations entre processus du même nœud ainsi que leur synchronisation dégradent les performances.

8. On sait seulement qu'il sera grand...

Interblocages La majorité des applications parallèles ne se décomposent pas en tâches indépendantes. Dans certains cas, les interactions entre ces tâches peuvent être complexes et mener au blocage temporaire de leur exécution. Typiquement, c'est le cas lorsqu'une tâche *mère* a besoin des résultats calculés par ses *filles* pour pouvoir poursuivre son exécution. Une bonne illustration de ce comportement est fournie par les applications se décomposant en *arborescence de tâches* dans lesquelles le calcul s'effectue en une phase d'expansion de l'arbre (création des tâches) suivie d'une phase de remontée des résultats. L'implantation parallèle du *tri-fusion* en est un bon exemple.

Le problème posé par ces applications réside dans le fait qu'un nombre potentiellement important de tâches peuvent être simultanément bloquées en attente de résultats. Cette caractéristique exige un effort de programmation particulier du traitement effectué par les processus serveurs de manière à empêcher leur blocage (faute de quoi un interblocage général de l'application pourrait survenir). Résoudre ce problème dans un cadre où le nombre de processus serveurs est fixe[9] revient à leur donner la possibilité de prendre en charge plusieurs tâches simultanément en passant de l'une à l'autre en cas de blocage, ce qui implique l'utilisation de mécanismes de sauvegarde et de restauration de contextes d'exécution. En fait, cette dernière approche consiste en l'introduction de multiple flots d'exécution au sein de chaque processus lourd, c'est-à-dire à l'utilisation d'un *parallélisme à grain fin* dans l'application.

Recouvrements des communications Lorsqu'un processus devient prêt à accepter la prise en charge d'une nouvelle tâche, il exécute une instruction `attendre_requête` (cf. code exécuté par les processus serveurs). Selon la politique de distribution des tâches employée, deux cas de figure sont possibles :

1. Le processus indique à l'environnement (par un envoi de message) qu'il est oisif, puis se bloque en attente d'une réception de message. Cette approche, si elle garantit une assez bonne distribution des tâches, peut occasionner une baisse d'efficacité importante si la durée des tâches est courte, en raison des pertes de *temps processeur* au cours des opérations d'attente de requêtes (aucune communication n'est recouverte par du calcul).
2. L'« environnement » (par exemple un processus centralisé distribuant la charge) anticipe les demandes de travail des processus en envoyant par avance une ou plusieurs requêtes à chacun d'eux, de manière à minimiser les risques de non-recouvrement des communications par le calcul. Cependant, cette approche augmente les risques de déséquilibre de charge en raison de sa distribution « en aveugle » d'une partie des tâches de l'application dont la durée — rappelons-le — n'est pas connue à l'avance. Lorsqu'une telle situation de déséquilibre survient, il est possible de procéder à un redistribution des tâches entre processus, mais à nouveau au prix de communications non-recouvertes.

On le voit, l'utilisation de configurations constituées d'un processus par nœud posent l'inévitable problème du recouvrement des communications par le calcul. Une solution consistant à utiliser plusieurs processus lourds par nœud pourrait être envisagée, mais elle occasionnerait une baisse de performances due aux commutations inter-processus

9. Ou même limité à un petit nombre.

imposées par le système d'exploitation. Nous verrons, dans le chapitre suivant, qu'une bonne solution réside (comme pour le point précédent) dans l'utilisation de multiples flots d'exécutions au sein des processus lourds.

Prise en compte des priorités Certaines applications parallèles peuvent se décomposer en tâches parallèles de *priorité* différente. C'est typiquement le cas pour les applications concernant l'exploration d'espaces de recherche guidée par heuristiques. Ce peut aussi être le cas lorsqu'une application peut prédire le caractère « générateur de parallélisme » de certaines de ses tâches, auquel cas il est souvent souhaitable de favoriser leur exécution. Pour ces applications, il est particulièrement important de veiller au respect des priorités des tâches lors de leur affectation sur les processeurs de la machine. Avec une configuration de processus serveurs telle que décrite précédemment, cette hypothèse se heurte à plusieurs difficultés :

- Pour assurer un recouvrement des communications par du calcul, les applications utilisent souvent une stratégie de distribution « anticipée » des tâches telle que décrite au point précédent. Cette distribution aveugle de certaines tâches peut mener à l'envoi d'une requête prioritaire vers un processus chargé auquel cas elle sera tamponnée par le système pendant une durée indéterminée.
- Même dans le cas où les requêtes ne sont pas envoyées par anticipation, il peut être gênant de laisser s'exécuter des requêtes non-prioritaires dans le système alors que des requêtes prioritaires sont en attente d'être traitées. Pour remédier à ce problème, il faudrait être capable, au sein d'un processus, d'interrompre l'exécution d'une requête au profit d'une autre. Cela nous amène à nouveau à la définition d'un mécanisme permettant de supporter plusieurs flots d'exécution différents au sein d'un même processus.

Force est donc de constater qu'en raison de ces problèmes, la gestion des priorités dans les environnements à gros grain n'est pas réalisable de façon satisfaisante.

2.3.1.3 Conclusion

Au vu des problèmes évoqués précédemment, il est facile de voir que les environnements de programmation parallèle à gros grain sont mal adaptés au support des applications irrégulières massivement parallèles. En fait, les processus, qui sont des entités gérées par le noyau du système d'exploitation, représentent des unités d'exécution dont la gestion est beaucoup trop *lourde* par rapport au grain de parallélisme potentiellement très fin des applications massivement parallèles. En particulier, des caractéristiques telles que les droits UNIX, la gestion des signaux ou encore la protection totale de leur espace d'adressage sont des caractéristiques qui n'ont pas d'intérêt particulier dans le cadre du calcul parallèle.

Au chapitre suivant, nous allons examiner un concept susceptible d'apporter une solution aux problèmes évoqués précédemment — le *parallélisme à grain fin* — et montrer dans quelle mesure son intégration dans un exécutif pour applications parallèles est intéressant.

Chapitre 3

Le parallélisme à grain fin

Ce chapitre a pour objectif de montrer l'intérêt du multithreading pour le support direct des applications irrégulières massivement parallèles. L'essentiel du chapitre est donc axé sur la description du concept de multithreading lui-même, c'est-à-dire sur la notion de *processus léger*, en insistant particulièrement sur les caractéristiques qui la distinguent radicalement de la notion de processus traditionnel.

Nous examinons d'abord les caractéristiques de base d'un processus léger, les différentes politiques d'ordonnancement les plus répandues et les possibilités de gestion de niveaux de priorités multiples. Ensuite, nous mentionnons quelques inconvénients notables liés à leur utilisation.

En conclusion, nous examinons dans quelle mesure ce concept permet de résoudre les problèmes évoqués à la fin du chapitre précédent, à savoir les inconvénients engendrés par l'utilisation d'environnements à gros grain pour le développement d'applications massivement parallèles irrégulières. Pour cela, nous commençons par récapituler les différentes catégories d'utilisations « courantes » des processus légers en insistant sur les avantages obtenus dans chaque cas de figure. Nous concluons ensuite en montrant les apports des processus légers au support d'applications irrégulières massivement parallèles.

3.1 Introduction

Le multithreading est une technique permettant la gestion de flots d'exécution multiples au sein des entités d'exécution de base fournies par un système d'exploitation. Ces flots d'exécution sont appelés des *processus légers* (*threads*), par opposition aux entités supportées par le noyau appelées *processus lourds*, ou plus simplement *processus*. La raison pour laquelle ces flots d'exécution sont appelés *processus légers* ainsi que les propriétés qui leur sont intrinsèques sont présentées dans ce qui suit.

Pour fixer les idées, tout au long de cette section nous nous placerons dans le contexte d'un système de type UNIX. Ainsi, nous parlerons parfois de *processus* UNIX pour désigner les processus lourds. Cette précision étant uniquement faite pour nous permettre d'illustrer notre propos à l'aide d'exemples concrets, celle-ci ne doit en aucun cas être considérée comme une restriction du modèle des processus légers au système UNIX.

3.2 Caractéristiques de base d'un processus léger

Comme nous venons de l'évoquer, un processus léger est un flot d'exécution interne à une entité lui servant de *contenant* : le processus lourd (figure 3.1). Un processus lourd peut ainsi contenir plusieurs centaines (voire même plusieurs milliers) de processus légers, qui constituent autant de flots d'exécution parallèles indépendants. Un processus lourd classique, c'est-à-dire ne contenant pas de processus léger, est dit *monoprogrammé*. Il ne contient qu'un seul flot d'exécution, qui exécute le code du processus de manière séquentielle, par exemple en exécutant la fonction *main* d'un programme C. Un processus lourd contenant un ou plusieurs processus légers est dit *multiprogrammé*. Dans ce cas, le flot d'exécution principal du processus est souvent considéré comme un processus léger particulier, dont le lancement est antérieur à tous les autres et dont la terminaison force la terminaison du processus lui-même (donc de tous les processus légers qu'il contient).

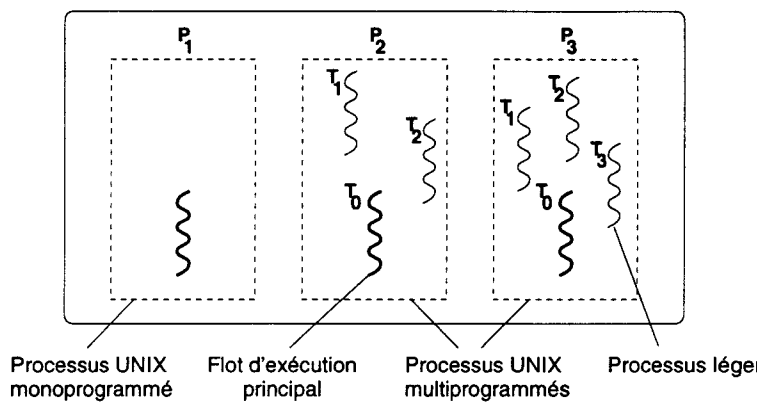

FIG. 3.1 - *Un processus* UNIX *peut contenir plusieurs flots d'exécution indépendants : les processus légers.*

Les processus légers d'un même processus lourd partagent un certain nombre de ressources. Le code est la principale de ces ressources. Ainsi, la création d'un processus léger supplémentaire ne conduit-elle jamais à la duplication de tout ou partie du code du processus lourd englobant. Pour autant, les processus légers constituent réellement des flots d'exécution indépendants, exécutant chacun une portion de code de façon non-synchronisée. Pour cela, chaque processus léger possède un *compteur ordinal* et une *pile d'exécution* qui lui sont propres. Ils constituent d'ailleurs l'essentiel de ses ressources. En effet, les ressources telles que le tas (mémoire utilisateur), la table des fichiers ouverts, ou encore la table des traitements de signaux sont également partagées par tous les processus légers d'un même processus lourd, qui ont en commun le même espace d'adressage (nous y reviendrons un peu plus loin).

En fait, un processus léger est dit « *léger* » justement parce qu'il possède très peu de ressources propres et que la taille de ces ressources est petite par rapport à celles utilisées par un processus lourd (la taille d'un processus léger se compte en kilo-octets, celle d'un processus en méga-octets). Ces caractéristiques expliquent en partie le fait qu'un processus léger soit une entité **beaucoup plus efficace** à gérer qu'un processus lourd.

La création d'un processus léger, par exemple, ne nécessite qu'un très petit nombre d'opérations : l'allocation d'un descripteur et d'une pile, l'initialisation du contexte d'exécution (affectation de registres) et l'insertion dans la file locale des processus prêts. La conséquence

est qu'une telle création est de 10 à 100 fois plus rapide qu'une création de processus lourd.

Le tableau 3.1 compare les temps d'exécution d'une création de processus lourd par rapport à une création de processus léger sur plusieurs architectures. Pour chaque architecture, la première ligne montre le temps d'exécution de la primitive de création (*Fork*), la deuxième montre le temps d'exécution depuis la création d'un processus n'effectuant aucune opération jusqu'à la récupération de sa terminaison par le système (*Fork & Wait*).

TAB. 3.1 - *Comparaison des opérations de création de processus lourd/léger (en millisecondes) sur deux architectures courantes.*

PC/Pentium 120MHz 48 Mo RAM sous Linux

Opération	Processus UNIX (Linux 2.0.17)	Processus léger (bibliothèque MARCEL)
Fork	0,660	0,018
Fork & Wait	1,211	0,034

Station Sun/Sparc5 70MHz 32 Mo RAM sous Solaris

Opération	Processus UNIX (Solaris 2.4)	Processus léger (bibliothèque MARCEL)
Fork	1,750	0,066
Fork & Wait	13,360	0,127

De la même manière, le changement de contexte, qui est l'opération déclenchée par l'ordonnanceur des processus légers pour effectuer une commutation du processeur d'un contexte d'exécution à un autre, ne nécessite que très peu d'opérations: seuls les registres du processeurs doivent être repositionnés. Encore une fois, les performances sont environ 10 à 100 fois meilleures que dans le cas d'un changement de contexte entre processus lourds.

Comme indiqué précédemment, la taille des ressources impliquées dans le fonctionnement d'un processus léger n'est pas le seul facteur expliquant les performances des primitives de gestion des processus légers. En observant la figure 3.2, qui montre la structuration de l'espace mémoire d'un processus lourd contenant des processus légers, on s'aperçoit que toutes les ressources propres à la gestion des processus légers se trouvent dans l'espace d'adressage *utilisateur*, c'est-à-dire qu'elles sont accessibles directement par le programme, sans nécessiter l'intervention du noyau.

C'est là une caractéristique différenciant radicalement un processus léger d'un processus lourd, puisqu'une partie de la structure de ce dernier est exclusivement gérée par le noyau du système (droits, table des fichiers, etc.). Cette organisation est ainsi faite de manière à ce que le noyau puisse avoir un contrôle total sur l'utilisation des différentes ressources de la machine (mémoire, périphériques, processeur, ...). Lorsqu'un processus désire accéder à une de ces ressources, il doit envoyer une requête au système qui vérifiera la validité de l'opération avant de l'effectuer réellement.

Cette organisation n'a pas d'équivalent au niveau des processus légers et chacun d'entre eux est libre d'utiliser les ressources du processus englobant directement, sans autre forme de contrôle que l'autodiscipline. Bien que ce mode de fonctionnement pose évidemment des problèmes de sécurité et d'intégrité des données partagées entre les processus (problèmes sur lesquels nous reviendrons par la suite), il a un avantage énorme: celui de permettre à un ordonnanceur de processus légers de ne jamais solliciter d'intervention du noyau pour effectuer

une commutation de contexte entre processus. C'est ce qui explique pourquoi une commutation de contexte entre deux processus légers est beaucoup plus rapide qu'une commutation de contexte entre deux processus lourds.

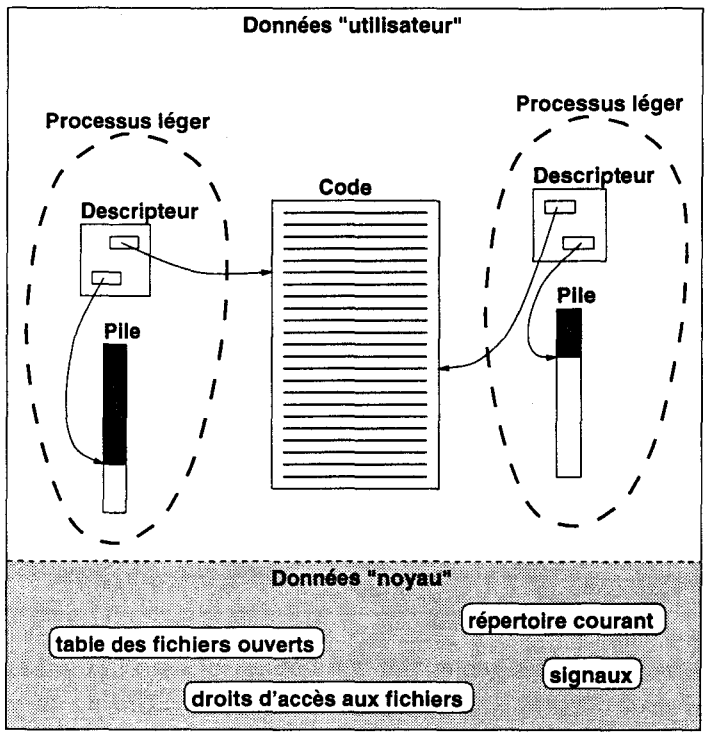

FIG. 3.2 - *Structure d'un processus lourd contenant deux processus légers.*

3.2.1 Limitations

Malgré les avantages que l'on peut obtenir grâce à leur utilisation, les processus légers que nous venons de décrire présentent un modèle de fonctionnement inadapté à certaines situations. Nous allons maintenant évoquer trois de ces situations et examiner précisément, pour chacune d'entre elles, le problème posé. Ensuite, nous introduirons une notion dérivée de celle de processus léger, celle de *processus de poids moyen*, qui corrige ces problèmes.

Protection mémoire Les bonnes performances associées à l'utilisation des processus légers ne constituent pas leur unique atout. Entre autres, la multiprogrammation est une façon souvent naturelle d'exprimer le parallélisme inhérent à une application. Dans ce cas, le principal intérêt d'utiliser des processus légers plutôt que des processus lourds est lié aux facilités qu'ils offrent en ce qui concerne le partage de mémoire. Alors que les processus lourds doivent, pour communiquer via une région mémoire commune, faire appel à un mécanisme d'allocation de segments de mémoire partagée [144], les processus légers, eux, n'ont qu'à accéder aux variables du processus englobant.

Cette forme d'utilisation des processus légers est plutôt guidée par des critères d'adéquation au support d'un parallélisme fin que par des critères de performances. Dans

ce contexte, l'absence de tout mécanisme de protection mémoire entre les processus légers peut s'avérer rédhibitoire. En effet, d'après ce que nous avons vu jusqu'à présent, rien n'interdit à un processus léger d'accéder en écriture aux données situées dans la pile d'un autre processus léger. Même si cette idée n'effleurera sans doute que très peu de programmeurs, les bogues de leurs programmes, en revanche, pourraient facilement combler cette lacune ! Ce problème, réel pour certaines applications, est dû à l'évolution des processus légers dans un espace d'adressage unique.

Parallélisme réel Un autre intérêt des processus légers pourrait être[1] de permettre une exploitation aisée et efficace des machines multiprocesseurs à mémoire commune. En effet, la structure d'un processus lourd contenant plusieurs processus légers coïncide parfaitement avec celle d'une machine à mémoire commune contenant plusieurs processeurs. Une application décomposée en n processus légers concurrents pourrait alors réellement s'exécuter en parallèle sur une machine à mémoire commune comportant au moins n processeurs.

Il y a malheureusement un problème : un ordonnanceur de processus légers s'exécutant en mode *utilisateur* (par opposition au mode *noyau*) n'a pas la possibilité d'accéder aux processeurs de la machine sous-jacente. Seul le noyau du système d'exploitation possède ce droit. Il résulte de ceci que le mécanisme des processus légers, du moins tel que nous l'avons présenté, n'est pas capable d'exploiter le parallélisme réel d'une machine. En fait, il ne peut introduire qu'un *pseudo-parallélisme* au sein d'un processus lourd, de la même façon qu'un ordonnanceur UNIX qui s'exécute sur une machine monoprocesseur.

Appels système bloquants De plus, compte tenu du fait qu'il est entièrement logé dans l'espace utilisateur, cet ordonnanceur n'est pas visible par le noyau. Ce dernier, en effet, ne « voit » qu'un processus UNIX classique et ignore totalement si oui ou non un ordonnanceur de processus légers officie à l'intérieur. Il n'existe donc pas d'interaction directe entre l'ordonnanceur UNIX et les différents ordonnanceurs « *légers* » qui peuvent s'exécuter dans certains processus du système. Si, du point de vue de l'efficacité, c'est une bonne chose (moins il y a d'appels au noyau, mieux c'est), cela pose problème d'un point de vue fonctionnel.

En particulier, dès qu'un processus léger effectue un appel à un service du noyau bloquant (par exemple une opération d'entrée/sortie), il risque le blocage du processus UNIX tout entier. C'est fort gênant si, à ce moment, d'autres processus légers (coexistant dans le même processus) sont éligibles pour être exécutés. Il est possible de remédier à ce problème de deux façons, mais aucune n'est réellement satisfaisante.

La première consiste tout simplement à s'abstenir d'effectuer des appels bloquants et d'utiliser à la place les primitives non-bloquantes correspondantes, de manière périodique. Par exemple, lorsqu'un processus léger désire lire un caractère depuis l'entrée

1. À cet endroit du discours, comme nous ne sommes pas censés savoir que c'est possible, le conditionnel s'impose...

standard, il peut exécuter le code suivant :

```
if je_ne_suis_pas_le_seul_processus_leger_actif
   then
      loop
        if caractere_disponible /* appel non-bloquant */
           then exit_loop
           else yield /* rend la main a l'ordonnanceur */
        end if
      end loop
end if
lire_caractere /* appel potentiellement bloquant */
```

Cela permet, tant que le caractère n'est pas disponible, de céder le processeur éventuellement à d'autres processus légers. Cette scrutation périodique, s'apparentant fortement à de l'attente active, peut avoir de lourdes conséquences sur les performances d'une application. De plus, ce mode de fonctionnement impose de réécrire toutes les primitives bloquantes de la bibliothèque standard du système, car on ne peut évidemment pas exiger d'un concepteur d'application de programmer explicitement cette scrutation (d'autant que la primitive *je_ne_suis_pas_le_seul_processus_léger_actif* fait rarement partie de l'interface de programmation des processus légers).

La deuxième façon de contourner ce problème n'est possible qu'avec un système d'exploitation fournissant une interface à des fonctionnalités d'*entrées/sorties asynchrones* [48]. Ces dernières permettent d'envoyer, de manière non-bloquante, une requête d'entrée/sortie au système, puis de continuer l'exécution de l'application. Une fois que la requête est effectivement *satisfaite*, le système envoie un *signal* au processus lourd pour lui notifier la fin de l'opération. Avec ce mécanisme, il est possible, pour un ordonnanceur de processus légers, de gérer correctement le blocage des processus légers, puisqu'il existe alors une interaction avec le noyau du système. Mais, rappelons-le, ces fonctionnalités ne sont pas disponibles sur tous les systèmes UNIX, ce qui limite la portabilité d'un ordonnanceur de processus légers basé sur ce principe. Notons enfin que, comme dans le cas précédent, ce mode de fonctionnement impose de réécrire toutes les primitives bloquantes de la bibliothèque standard du système.

Notons que sur des systèmes spécialisés [2, 6, 114], des protocoles de communication entre une application et le noyau sont fournis et permettent par exemple à un ordonnanceur de processus légers de récupérer un signal en cas d'appel bloquant. Cette fonctionnalité, qui revient à peu près à fournir des mécanismes d'entrées/sorties asynchrones, n'est cependant pas disponible sur tous les systèmes UNIX.

C'est principalement pour remédier à ces problèmes que les concepteurs de certains systèmes d'exploitation (tels que Solaris [157]) ont proposé une notion de processus intermédiaire entre les processus lourds et les processus légers : les *processus de poids moyen* (*medium-weight processes*).

3.2.2 Les processus de poids moyen

Un processus de poids moyen, encore appelé processus léger *géré par le noyau*, possède les mêmes caractéristiques fonctionnelles qu'un processus léger. En particulier, il s'exécute

au sein d'un processus lourd et en partage le code ainsi que les variables globales avec les autres processus de poids moyen du même processus. Cependant, et c'est là sa différence fondamentale avec un processus léger, son ordonnancement est effectué au niveau **du noyau du système.**

Concrètement, cela signifie que les opérations de création, destruction, commutation ou encore de synchronisation provoquent une intervention du noyau à chaque fois qu'elles sont sollicitées. De plus, les structures internes décrivant chacun de ces processus (les *descripteurs*) se trouvent dans l'espace d'adressage du noyau (figure 3.3), comme le sont celles décrivant les processus lourds. De cette manière, le système est parfaitement capable de gérer correctement les appels systèmes bloquants, puisqu'il connaît précisément l'identité du processus « léger » effectuant l'appel.

De plus, puisque c'est le noyau du système qui prend en charge les opérations de commutation entre processus (de poids moyen), il lui est possible d'attribuer à chacun de ces processus un espace d'adressage privé (pour le stockage de la pile d'exécution ainsi que de données spécifiques au processus) qu'il lui suffit de sauver/restaurer à chaque changement de contexte.

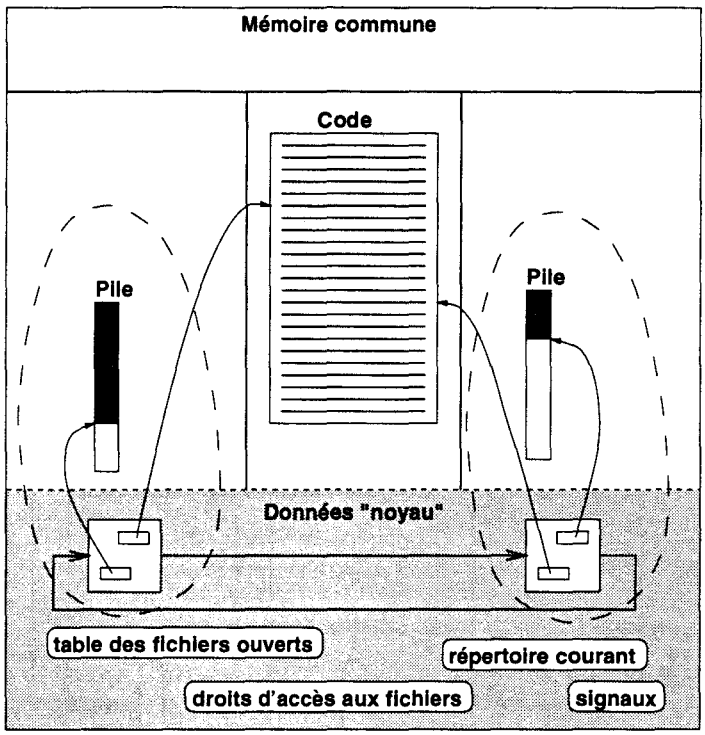

FIG. 3.3 - *Structure d'un processus lourd contenant deux processus légers* **gérés par le noyau.**

Bien évidemment, le problème posé par l'exploitation du parallélisme réel fourni par les machines multiprocesseurs à mémoire commune ne se pose plus avec les processus de poids moyen. En effet, puisque c'est le système qui prend en charge l'ordonnancement de ces processus, il peut tout à fait placer l'exécution de processus de poids moyen sur des processeurs différents, de la même manière qu'il le ferait avec des processus lourds. Nous reviendrons sur

ce point dans la section consacrée aux différentes politiques d'ordonnancement des processus légers.

Toutes ces bonnes propriétés, intrinsèques aux processus de poids moyen, ont une contrepartie logique : les performances des primitives associées à leur gestion sont sensiblement inférieures à celles associées à la gestion des processus légers. C'est l'illustration parfaite du difficile compromis que réalise la gestion des processus d'un système : « *ce que l'on gagne en fonctionnalités, on le perd en efficacité, et vice versa* ». En ce sens, les processus de poids moyen représentent véritablement un concept intermédiaire entre le concept de processus lourd (fonctionnalités) et le concept de processus léger (efficacité).

Malgré l'intéressant compromis qu'ils représentent, les processus de poids moyen ne sauraient toutefois se substituer aux processus légers en n'importe quelle circonstance. En particulier, sur une machine monoprocesseur, une application comportant un degré de parallélisme massif (beaucoup de processus légers potentiels) et, par exemple, de fréquents points de synchronisation, perdrait beaucoup d'efficacité en utilisant des processus de poids moyen au lieu des processus légers.

C'est sans doute la raison pour laquelle certains systèmes d'exploitation, notamment Solaris [109], supportent les deux concepts. Cela introduit alors deux niveaux d'ordonnancement dans le système [141] : le niveau noyau, qui prend en charge l'ordonnancement des processus de poids moyen, et le niveau utilisateur (*i.e.* s'exécutant en mode utilisateur) qui prend en charge l'ordonnancement des processus légers [151].

3.2.2.1 Exemple : le système UNIX Solaris

Dans le système Solaris, l'interaction entre ces deux niveaux est assez complexe et permet en outre d'effectuer l'ordonnancement de plusieurs processus légers *au-dessus* d'un groupe de processus de poids moyen (figure 3.4). Ces derniers, qui sont les entités actives de base du système, représentent alors de véritables *processeurs virtuels* pour les processus légers qu'ils supportent.

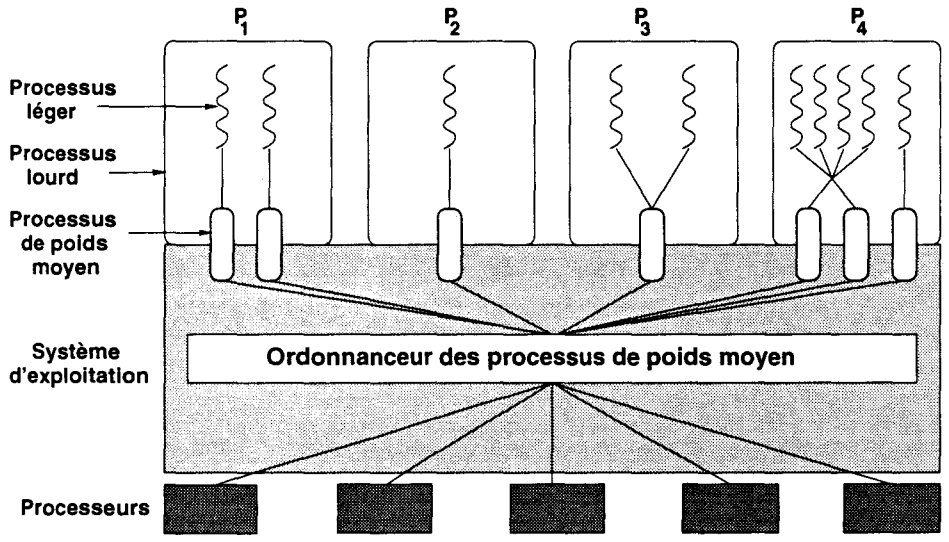

FIG. 3.4 - *Le modèle de processus légers à deux niveaux du système Solaris.*

La figure 3.4 illustre plusieurs cas de figure possibles sur un tel système. Le processus P_2 représente un processus UNIX au sens classique du terme, c'est-à-dire ne contenant qu'un seul flot d'exécution. Dans ce cas, l'exécution de ce processus est directement supportée par un processus de poids moyen et tout se passe comme si le processus contenait un seul processus léger.

Le processus P_3 représente la structure par défaut d'un processus multiprogrammé contenant deux processus légers. Dans ce cas, l'exécution est supportée par un seul processus de poids moyen et un ordonnanceur de niveau utilisateur assure les commutations entre processus légers. Tout se passe exactement de la même manière que sur un système traditionnel avec une bibliothèque de processus légers classique. Si l'un de ces deux processus effectue un appel système bloquant, c'est directement le processus de poids moyen qui passe dans l'état bloqué[2], entraînant du même coup le blocage du deuxième processus léger. En fait, cela n'est que temporaire, car dans ce cas le système réagit en créant un nouveau processus de poids moyen pour continuer l'exécution du deuxième processus léger.

La structure du processus P_1 présente un cas de figure intéressant. Ce processus renferme, comme le processus P_3, deux processus légers. Cependant, dans le cas présent, il a été demandé au système de lier un processus de poids moyen au support de chacun de ces processus légers (*bound threads*). C'est l'exemple typique d'une application utilisant au maximum les fonctionnalités des processus de poids moyen. En particulier, le blocage d'un des deux processus légers n'entraîne pas directement le blocage de l'autre. De plus, un parallélisme réel peut être obtenu au sein du processus lourd.

Enfin, le processus P_4 est représentatif de la flexibilité de l'ordonnancement à deux niveaux du système Solaris. Il contient d'une part un processus léger dont l'exécution est liée à un processus de poids moyen, et d'autre part un ensemble de quatre processus légers pris en charge par un *pool* de processus de poids moyen. Cela signifie qu'au maximum deux des quatre processus légers seront réellement exécutés en parallèle. Par rapport au mécanisme de liaison entre un processus léger et un processus de poids moyen, cette dernière possibilité permet de fixer le degré maximum de parallélisme réel (*concurrency level* dans la terminologie Solaris) intra-application de façon à limiter la consommation des ressources système.

Notons qu'il existe de nombreuses variantes et fonctionnalités additionnelles à ces mécanismes, mais leur énumération exhaustive sortirait du cadre de notre étude.

3.3 Politiques d'ordonnancement

Les principaux concepts de base des processus légers étant posés, nous allons maintenant examiner quelques politiques d'ordonnancement communément utilisées dans les différentes bibliothèques de multiprogrammation existantes. Nous nous focaliserons exclusivement ici sur le problème de l'ordonnancement sur une machine monoprocesseur, car l'ordonnancement sur une machine multiprocesseur nécessite soit une intégration dans le noyau du système, soit l'utilisation de processus de poids moyen. Nous reviendrons sur cette dernière possibilité dans la section *Perpectives de ce travail* (plus précisément sur le portage du noyau MARCEL sur machines multiprocesseurs).

Un ordonnanceur est un programme chargé d'arbitrer l'accès au processeur des différents processus éligibles à un instant donné. Lorsqu'un programme utilisant une bibliothèque de

2. étant donné que seul le processus de poids moyen est géré au niveau du noyau.

processus légers s'exécute, il sollicite l'intervention de deux ordonnanceurs : celui du système et celui de la bibliothèque de processus légers (figure 3.5).

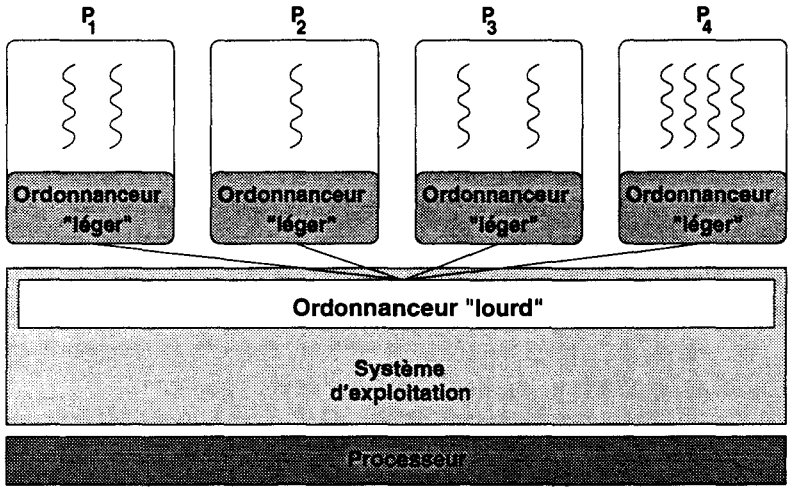

FIG. 3.5 - *Chaque processus lourd possède son propre ordonnanceur de processus légers.*

L'ordonnanceur du système (par exemple UNIX) prend en charge l'attribution du processeur aux différents processus lourds s'exécutant sur la machine. Les différents types de stratégies d'ordonnancement dans les systèmes d'exploitation sont légion et varient beaucoup en fonction du type d'applications visées (applications interactives, traitements par lots, ...) [158]. Nous ne détaillerons pas ces stratégies ici, car nous ne parlerons par la suite que de processus (lourds) s'exécutant « seuls » sur un processeur. Dans ce cas, nous considérerons que la totalité du processeur sera attribuée à un tel processus, en négligeant les ressources consommées par les processus système s'exécutant en tâches de fond.

L'ordonnanceur de la bibliothèque de processus légers, quant à lui, prend en charge le découpage en tranches du temps alloué au processus lourd et leur attribution aux processus légers. Comme nous supposons que le processus s'exécute seul sur le processeur, nous pouvons alors considérer que cet ordonnanceur régit tout simplement le partage du processeur entre les différents processus légers.

Nous allons maintenant analyser les différentes politiques d'ordonnancement les plus répandues dans les ordonnanceurs de processus légers.

3.3.1 Exécution sans interruption

Certains systèmes ou environnements de programmation parallèle proposent une restriction de la notion de processus léger à celle de *fil d'exécution non-interruptible*. Dans de tels systèmes, l'exécution des processus légers est toujours effectuée en une seule fois, jusqu'à la terminaison du processus. C'est par exemple le cas des *threads* du système Filaments [62], ou encore de l'environnement Cilk [13].

Le code exécuté par ces processus est évidemment soumis à de larges restrictions, puisqu'il ne peut effectuer d'appel ni à une primitive système bloquante ni même à une éventuelle primitive de synchronisation de la bibliothèque. Le principe général de programmation est de découper l'exécution d'un processus léger comportant des points de synchronisation en une

séquence de « filaments » (ne comportant aucun point de synchronisation) dont la création sera conditionnée par le point de synchronisation les précédant. Ce mécanisme est illustré sur la figure 3.6.

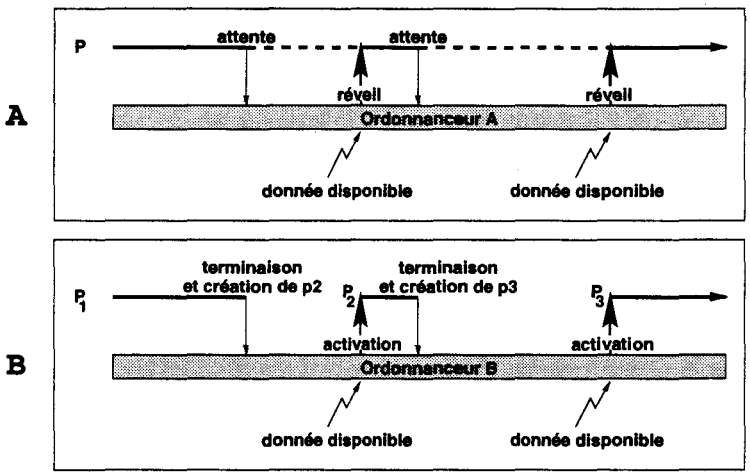

FIG. 3.6 - *La programmation à l'aide de filaments (B), opposée à la programmation à l'aide de processus légers classiques (A).*

Le cas de figure *A* montre l'exécution d'un processus léger classique effectuant à deux reprises un appel bloquant (attente de données). Le cas de figure *B* illustre ce qui se passe dans un système basé sur les filaments. Les appels bloquants sont remplacés par des créations de nouveaux filaments dont l'*activation* est conditionnée par un certain nombre de *conditions d'activation*, gérées par le noyau. Lorsque les conditions d'activation d'un filament sont toutes remplies, alors le filament peut être activé (exécuté entièrement).

Le travail de l'ordonnanceur d'un tel système (cf. l'ordonnanceur *B*) consiste donc principalement en la gestion d'une table des dépendances pour les processus en attente d'exécution.

Bien que plus difficile et moins pratique à utiliser que le concept de processus léger, le concept des filaments est intéressant à plusieurs titres.

D'abord, l'implantation d'un tel mécanisme est assez indépendante du système sous-jacent (donc portable), car il n'est nul besoin de mettre en place une gestion multi-contextes comme c'est le cas pour une implantation des processus légers. De plus, étant donné que les informations caractérisant un filament non encore activé sont indépendantes de la machine (liste de dépendances), l'implantation de mécanismes de migration de filaments (par exemple le *Work Stealing* dans Cilk [14]) entre différentes machines au sein de régulateurs de charges est facilement réalisable.

En outre, puisque son exécution n'est jamais interrompue, un filament n'a pas besoin non plus d'une pile d'exécution privée. En fait, les filaments sont des entités encore plus légères que des processus légers. Par conséquent, les mécanismes qui leur sont associés sont encore plus efficaces que ceux associés aux processus légers.

Enfin, étant donné que les filaments n'effectuent pas d'appels systèmes bloquants, l'ordonnanceur a une maîtrise totale sur le déroulement des programmes, ce qui permet la mise en place de mécanismes complexes tels que la détection de famine par exemple.

Malgré ces qualités, le manque d'universalité du concept de filament (*i.e.* il est destiné à

des applications s'exprimant dans un mode « DataFlow ») fait qu'il n'est pas très utilisé dans la communauté des concepteurs d'applications parallèles.

3.3.2 Exécution préemptible

Contrairement au modèle d'exécution des filaments, celui des processus légers permet à un processus léger de passer de l'état *prêt* ou *actif* à l'état *bloqué* et vice versa. À tout moment dans le système coexistent donc un ensemble de processus prêts et un ensemble de processus bloqués. Parmi l'ensemble des processus légers prêts, un seul peut être actif à la fois (machine monoprocesseur). La figure 3.7 illustre les transitions possibles entre les différents états d'un processus léger.

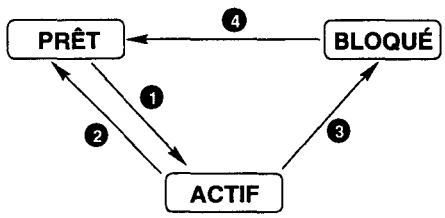

FIG. 3.7 - *Graphe du changement d'état des processus légers.*

C'est le rôle de l'ordonnanceur d'assurer ces différents changements d'états pour l'ensemble des processus légers s'exécutant dans un processus. Ces transitions signifient :

1. L'acquisition du processeur pour un processus léger éligible à l'exécution. Ce processus est choisi par l'ordonnanceur parmi l'ensemble des processus prêts.

2. La libération du processeur par le processus léger actif, provoquée par une réquisition de l'ordonnanceur ou par l'exécution d'une instruction de commutation explicite.

3. La libération du processeur par le processus léger actif, provoquée par un blocage de ce dernier (par exemple pendant une opération d'entrée/sortie).

4. Le passage à l'état prêt d'un processus léger bloqué, provoqué par une instruction de « *réveil* » du processus. Cette instruction peut avoir été exécutée par le noyau du système (pour notifier la disponibilité de certaines ressources) ou par le processus léger actif (primitive de synchronisation).

En fait, les transitions 3 et 4, qui concernent le blocage et le réveil des processus, sont, de par leur nature, prises en charge de la même façon par tous les ordonnanceurs de processus légers. Leur étude ne présente donc pas d'intérêt particulier ici. En revanche, les transitions 1 et 2 constituent véritablement le cœur du fonctionnement de l'ordonnanceur, car c'est sur leur implantation que repose la sémantique d'une grande partie des fonctionnalités des processus légers. Bien qu'il existe de multiples stratégies d'ordonnancement possibles, celles-ci sont classifiables en deux catégories, que nous allons maintenant examiner.

3.3.2.1 Exécution préemptible uniquement aux points de synchronisation

La première catégorie regroupe un ensemble de politiques d'ordonnancement souvent appelées, par abus, « *non-préemptives* ». En réalité, l'expression correcte serait « *non-préemptives en dehors des points de synchronisation* »[3]. En effet, une telle politique d'ordonnancement ne provoque l'interruption du processus léger en cours d'exécution (*actif*) que dans les situations suivantes :

- le processus actif arrive au terme de son exécution, ce qui provoque sa disparition ;
- le processus actif exécute une instruction de commutation explicite (*yield*) vers un autre processus (son état passe alors de `actif` à `prêt`) ;
- le processus actif exécute une primitive de synchronisation (par exemple l'attente sur un sémaphore) qui le bloque (son état passe alors de `actif` à `bloqué`) ;
- le processus actif exécute une primitive conduisant (directement ou indirectement) à la présence, parmi l'ensemble des processus prêts, de un ou plusieurs processus dont l'exécution est devenue plus *prioritaire* que la sienne (son état passe alors de `actif` à `prêt`).

Nous reviendrons sur le dernier point dans la section consacrée à la *gestion des priorités*.

Il découle des règles précédentes qu'un ordonnanceur non-préemptif réduit le nombre de commutations entre processus légers à son strict minimum. Cette propriété assure que l'exécution d'un processus léger n'est jamais interrompue au milieu d'un calcul, par exemple. Cette caractéristique fait de ce mode de fonctionnement un modèle proche de celui des coroutines [37], c'est pourquoi l'on parle parfois d'ordonnancement *coopératif*. Les principaux avantages de cette approche sont les suivants :

- Le temps passé par le système à effectuer les commutations (on parle aussi de *changements de contexte d'exécution*) entre processus est minimal et donc l'ordonnancement est performant. Ce facteur est important car une commutation entre deux processus nécessite **au minimum** la sauvegarde de l'état de la machine (processeur, co-processeur arithmétique) vers la pile du processus relâchant le processeur, puis une restauration de cet état depuis la pile du processus acquérant le processeur. De fréquents déclenchements de cette opération, qui manipule beaucoup de registres sur certaines architectures (RISC), risqueraient donc d'introduire un *surcoût* significatif des opérations de l'ordonnanceur sur le déroulement normal du programme.
- Étant donné que l'ordonnanceur n'interrompt jamais un processus en dehors de points de synchronisation bien précis (appels à des primitives de bibliothèque de multiprogrammation), l'accès des processus aux données partagées s'effectue de manière exclusive entre deux points de synchronisation. Cette caractéristique simplifie grandement la programmation de l'accès aux ressources situées en mémoire globale, car les problèmes inhérents aux accès concurrents sont, dans leur majorité, évités.

  Par exemple, la simple instruction `x := x+1`, si elle est exécutée *concurremment* par deux processus légers, n'incrémente pas forcément la variable globale x deux fois au bout du compte. En effet, les deux processus peuvent chacun, dans un premier temps,

3. Mais c'est plus long...

charger la valeur de x dans leur version de l'accumulateur du processeur, puis dans un deuxième temps effectuer l'addition et enfin dans un troisième temps stocker le résultat à l'emplacement mémoire de x, ce qui aurait pour effet de n'avoir incrémenté la variable que de 1. Pour remédier à cela, il faut utiliser des outils de synchronisation permettant aux processus d'exécuter cette instruction de manière exclusive.

Avec un ordonnancement non-préemptif, toutefois, la situation précédente ne peut pas se produire et le recours aux outils de synchronisation est donc inutile. Par conséquent, la programmation des applications s'en trouve facilitée et du même coup leur efficacité s'en trouve améliorée.

- L'implantation d'un ordonnanceur non-préemptif est relativement facile, puisque celui-ci n'a jamais besoin de réquisitionner le processeur. Au contraire, c'est le processus actif qui, par l'exécution d'une primitive particulière, sollicite son intervention.

Malheureusement, l'ordonnancement non-préemptif n'a pas que des avantages et voici précisément ses inconvénients les plus notoires :

- L'accès des processus au processeur n'est pas équitable, car un processus effectuant un gros calcul conserve la « main » pendant toute la durée du calcul. Celui-ci est alors fortement avantagé par rapport à un processus effectuant la même quantité de travail, mais exécutant un code entrecoupé de points de synchronisation.

- Une première conséquence de ce manque d'équité est que ce type d'ordonnancement n'est pas du tout adapté aux applications dites « interactives », c'est-à-dire les applications dont on espère des temps de réponse courts lorsqu'elles sont sollicitées. Par exemple, une application connectée à une interface graphique doit, pour être effectivement utilisable, être capable d'exécuter rapidement un traitement associé à un bouton de l'interface, et cela concurremment à d'éventuelles activités s'exécutant en « tâches de fond ». Avec un ordonnancement non-préemptif, il n'est pas possible de garantir une telle propriété, puisqu'une activité peut monopoliser le processeur pendant un temps non-prévisible.

- La deuxième conséquence est qu'il peut y avoir des situations de *famine*, c'est-à-dire des processus prêts n'accédant jamais au processeur. C'est en particulier le cas si le processus actif exécute une boucle infinie ne contenant pas de point de synchronisation.

- Enfin, la mise en place d'un mécanisme de priorités d'exécution pour les processus légers est possible, mais celui-ci, en l'absence de préemption régulière, reste d'un intérêt très limité. Nous détaillerons ce point dans la section « *Gestion des priorités* ».

Malgré ces inconvénients, la politique d'ordonnancement non-préemptive demeure la plus utilisée dans les différentes bibliothèques de processus légers existantes. Les deux raisons principales sont 1°) la non-nécessité de protéger tous les accès aux ressources partagées et 2°) la facilité d'implantation d'un tel ordonnanceur.

Le mécanisme d'ordonnancement le plus répandu dans les bibliothèques non-préemptives est appelé *ordonnancement FIFO*, en raison de l'utilisation d'une file de processus prêts. Le principe général[4] de fonctionnement est très simple (figure 3.8).

4. De multiples variantes existent, mais le principe exposé ici reste valable

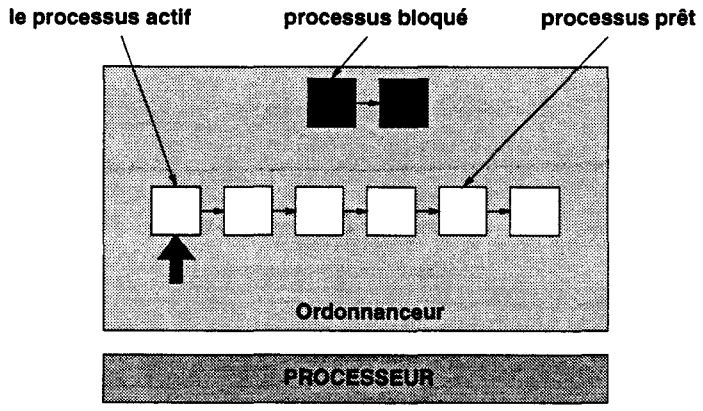

FIG. 3.8 - *Ordonnancement FIFO.*

L'ordonnanceur maintient une file contenant tous les processus *prêts* à s'exécuter. Le premier processus de cette file représente toujours le processus en cours d'exécution (le processus *actif*). Lorsque ce processus exécute une instruction provoquant son interruption (commutation explicite ou synchronisation bloquante), l'ordonnanceur prend le contrôle de l'exécution. Le processus est alors retiré de la tête de la file, puis éventuellement replacé en queue[5] s'il est toujours dans l'état prêt. Enfin, l'ordonnanceur effectue une commutation de contexte vers le processus se trouvant en tête de file.

3.3.2.2 Exécution préemptible en temps partagé

La deuxième grande catégorie de politiques d'ordonnancement regroupe les politiques appelées « *politiques préemptives* ». Là encore, il s'agit d'un abus de langage et l'expression appropriée est *politiques préemptives avec partage de temps*.

Un ordonnanceur de ce type comporte un mécanisme, s'apparentant à une horloge, qui provoque des changements de contexte à intervalles réguliers. Un tel intervalle est appelé un *quantum de temps* et représente typiquement quelques millisecondes. Un ordonnanceur préemptif garantit donc qu'un processus ne monopolise jamais consécutivement le processus pendant une durée supérieure à un quantum de temps. Bien évidemment, un processus peut n'utiliser consécutivement le processeur que pendant une durée inférieure à ce quantum, car des changements de contexte peuvent également se produire sur des points de synchronisation, de la même manière qu'avec un ordonnanceur non-préemptif.

Du point de vue des avantages et inconvénients d'utilisation, un ordonnanceur préemptif représente une solution contraire à un ordonnanceur non-préemptif. En particulier, les avantages sont :

- Étant donné qu'une horloge réglemente la durée maximale d'occupation consécutive du processeur, chaque processus voit son exécution découpée en « tranches » pour être entrelacée avec celle des autres (figure 3.9). Par conséquent, en l'absence de mécanisme de gestion de priorités, la répartition du temps processeur parmi les processus légers est uniforme, donc *équitable*[6].

5. Pour l'instant, nous nous plaçons dans un contexte où tous les processus ont la même priorité d'exécution.
6. En supposant que la stratégie de choix du processus actif parmi les processus prêts est équitable, bien

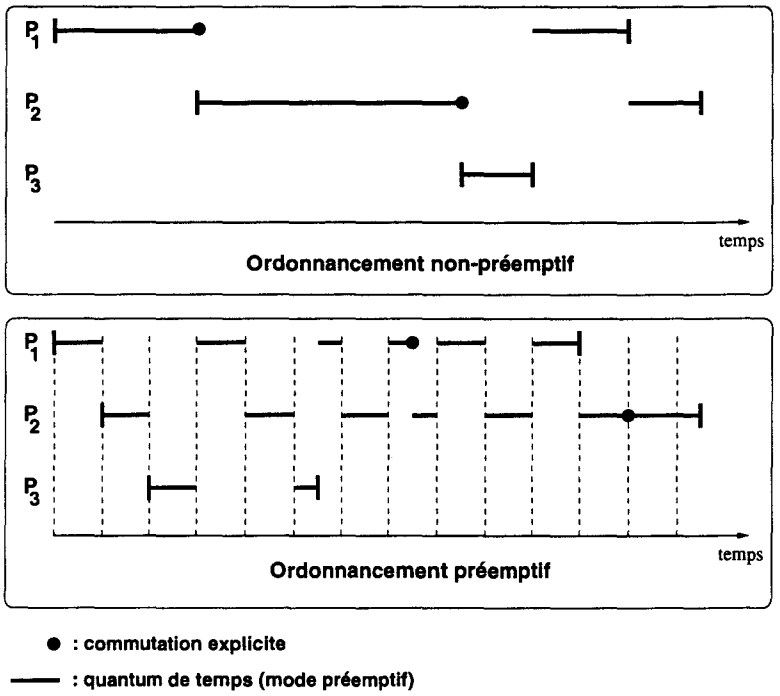

FIG. 3.9 - *Avec un ordonnancement non-préemptif, seuls les points de synchronisation provoquent des commutations. Avec un ordonnancement préemptif, une horloge entrelace les exécutions.*

- Le temps pendant lequel un processus léger prêt « attend » l'accès au processeur est borné par le produit du nombre de processus prêts par la durée d'un quantum de temps[7]. L'intérêt est que cette borne ne dépend pas de la quantité de travail des processus prêts. Ainsi, lorsqu'un nouveau processus est créé dans le système, il est possible de calculer une date au-delà de laquelle il aura forcément débuté son exécution. Le système est alors réellement interactif. Cette caractéristique est importante pour les applications comportant par exemple des processus légers chargés de « réagir » rapidement à certains événements. Nous verrons, dans la partie *Parallélisme à grain fin en milieu distribué*, l'importance de cette caractéristique dans un environnement distribué.

- L'entrelacement des exécutions ajouté à l'équité de l'ordonnancement fait que, hormis dans certains programmes très particuliers (mais heureusement irréalistes : cascades infinies de créations, ...), il ne se produit jamais de situation de famine. En effet, même en présence d'un processus léger exécutant une boucle infinie, un processus prêt accédera toujours au processeur au bout d'un temps fini.

- En disposant d'un mécanisme de préemption en temps partagé, capable de réquisitionner le processeur à tout moment de manière autoritaire, il est possible d'instaurer une politique d'ordonnancement basée sur la notion de priorité d'exécution. C'est ce que nous allons étudier dans la section suivante.

---

entendu.

7. En supposant un ordonnancement équitable

Alors que les avantages d'une politique non-préemptive concernent essentiellement des aspects d'efficacité et de facilité de mise en œuvre, nous constatons ici que les avantages d'une politique **préemptive** sont clairement liés aux qualités intrinsèques de l'ordonnancement. Bien entendu, l'obtention de ces qualités s'effectue au détriment de la facilité de mise en œuvre et de l'efficacité « brute » de l'ordonnancement :

- Nous avons évoqué dans la section précédente la quantité de travail minimale effectuée lors d'un changement de contexte entre deux processus. Dans un mode de préemption en temps partagé, il faut y ajouter les opérations effectuées par le système pour interrompre le processus courant lors d'un « top d'horloge ». Ces opérations, consistant souvent en l'envoi d'un *signal* (au sens UNIX) à l'ordonnanceur, occasionnent un surcoût notable par rapport à un simple mécanisme de commutation explicite.

- La perturbation occasionnée par les changements de contexte dus à la « préemption automatique » dépend de la fréquence à laquelle ils sont déclenchés, c'est-à-dire de la durée des *quanta* de temps instaurés par l'ordonnanceur. Généralement de l'ordre de quelques dizaines de millisecondes, ces quanta sont parfois paramétrables par l'application. Des quanta courts favorisent l'interactivité de l'application mais occasionnent une perturbation plus importante due aux changements de contexte très fréquents. Au contraire, des quanta longs atténuent la perturbation occasionnée par les commutations entre processus, mais diminuent l'interactivité de l'application. Toute la difficulté réside dans le choix de cette période voire même dans l'identification de ce que représente un quantum court (resp. long) sur un processeur particulier, car c'est effectivement en fonction de la puissance de ce dernier qu'il est nécessaire de « régler » ce paramètre.

- Étant donné le caractère fortement asynchrone des changements de contexte provoqués par l'horloge, ceux-ci représentent une réelle rupture dans le fonctionnement du programme. De ce fait, ils peuvent altérer, par exemple en perturbant le contenu des caches internes du processeur, les performances d'une application. Il convient cependant de nuancer cet argument, et cela pour au moins deux raisons. La première est qu'avec un système d'exploitation multiprogrammé tel que UNIX, de telles « ruptures de fonctionnement » sont nombreuses pendant la durée d'une application et sont dues à la réquisition du processeur par le noyau du système. La seconde est qu'un certain nombre d'expérimentations concernant l'influence de la préemption sur les effets de cache ont été menées (notamment par Christian Perez au LIP [139] et par Jean-François Méhaut au LIFL) et montrent des résultats contraires à certaines « intuitions » dans ce domaine. Nous y reviendrons dans le chapitre *Réalisation et Performances* (section 6.3.7).

- Le comportement asynchrone des commutations de contexte rend délicats les accès des processus aux ressources partagées. Pour reprendre un exemple précédemment cité, il n'est pas garanti qu'une instruction telle que `x := x+1` (x étant une variable globale) exécutée par plusieurs processus légers potentiellement concurrents ait effectivement pour conséquence l'incrémentation multiple de la variable x [8]. Pour résoudre ce problème de cohésion des ressources globales, il faut protéger leur accès à l'aide de mécanismes de synchronisation, par exemple avec des verrous (*mutex* dans la terminologie anglo-saxonne). Cette opération est loin d'être triviale, pour au moins trois raisons.

8. L'explication en est donnée dans la section précédente.

Tout d'abord, il n'est pas toujours possible d'effectuer une simple protection du code de façon à assurer son exécution en exclusion mutuelle, soit parce que l'exclusion mutuelle n'est pas nécessaire et que son utilisation risque de sérialiser beaucoup trop d'opérations, soit parce que le code contient des points de synchronisation ou des appels bloquants, ce qui amènerait alors des risques d'étreintes fatales (*deadlocks*) entre les processus.

Ensuite, le code en question peut être dispersé dans différents modules, eux-mêmes développés par différentes personnes, ce qui rend sa « sécurisation » très complexe et fastidieuse à réaliser.

Enfin, le code peut ne pas être disponible du tout, comme c'est le cas pour toutes les fonctions de la bibliothèque C standard des systèmes commerciaux. Les primitives d'allocations mémoire (*malloc*, *free*, etc.) en sont le parfait exemple, puisqu'elles accèdent à une ressource globale qui est la mémoire de tas (*heap*). Comme il n'est pas possible de modifier directement le corps de ces fonctions, il faut donc utiliser des fonctions intermédiaires ajoutant de la synchronisation autour de l'appel de base. Notons que lorsque la bibliothèque des processus légers est intégrée au système d'exploitation (comme dans le système Solaris), des versions sécurisées des primitives de la bibliothèque standard vis-à-vis des processus légers sont fournies. Ces primitives sont alors indiquées être *MT-Safe* (Multi-Thread Safe) dans la documentation du système.

La mise en œuvre d'un ordonnanceur préemptif, bien que plus délicate à réaliser que celle d'un ordonnanceur non-préemptif (principalement à cause de la gestion des interruptions de l'horloge), s'articule souvent autour des mêmes structures de données que ce dernier.

L'algorithme d'ordonnancement préemptif le plus répandu dans les systèmes à base de processus légers est l'algorithme du *tourniquet* (*round robin scheduler*, figure 3.10). Il est nommé ainsi en raison de l'utilisation d'un véritable « tourniquet » de processus prêts. Ce tourniquet, mû par l'horloge, tourne d'un cran (d'un processus) à chaque impulsion d'horloge. Une position fixe particulière, placée devant le tourniquet, désigne le processus léger en cours d'exécution. Ainsi, les processus prêts occupent tour à tour le processeur. Si le processus actif se bloque, se termine, ou exécute une instruction de commutation explicite, alors le tourniquet tourne d'un cran sans attendre la prochaine interruption d'horloge.

Cet algorithme a pour principales qualités d'être simple à mettre en œuvre, d'être efficace (car la recherche du prochain processus prêt est immédiate) et de posséder des propriétés d'ordonnancement faciles à prouver (durée maximale d'attente, etc.).

Cependant, comme l'ordonnancement FIFO (avec lequel il présente d'ailleurs beaucoup de similitudes), il suppose que les exécutions de chacun des processus sont d'égale importance. Or, cette dernière propriété n'est pas vraie pour toutes les applications. En particulier, il peut s'avérer nécessaire (ou parfois simplement souhaitable) d'attribuer des *priorités* distinctes à différents processus d'une application. C'est ce que nous allons voir maintenant.

3.4 Ordonnancement avec priorités

Pour certaines classes d'applications multiprogrammées, la gestion de multiples niveaux de priorités d'exécution est une nécessité. Par exemple, une application de calcul peut fournir une interface graphique permettant à un utilisateur d'agir interactivement sur son déroulement global (interrogation, déclenchement de nouveaux traitements, etc.). Dans ce cas, il est nécessaire que les actions déclenchées par l'utilisateur via l'interface soient prioritaires par rapport

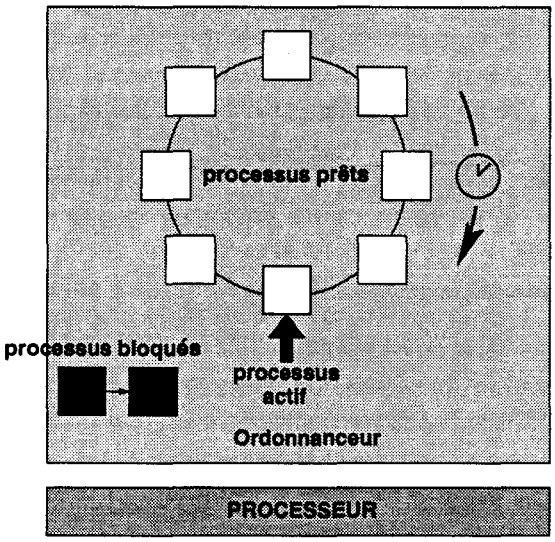

FIG. 3.10 - *Ordonnancement en tourniquet.*

au calcul s'effectuant en « tâches de fond ». C'est pourquoi la plupart des ordonnanceurs de processus légers, à l'instar de leurs aînés les ordonnanceurs des systèmes d'exploitation, intègrent la notion de priorité au cœur de leur algorithme d'ordonnancement.

En raison de la multitude des systèmes d'exploitation existants et de la diversité de leurs domaines d'utilisation, de très nombreuses politiques d'ordonnancement avec priorité ont vu le jour depuis l'origine des systèmes multiprogrammés [158]. Il n'en est cependant pas tout à fait de même en ce qui concerne les ordonnanceurs de processus légers puisque, en dehors des ordonnanceurs pour machines multiprocesseurs dont nous ne parlerons pas ici, peu de politiques d'ordonnancement différentes sont réellement utilisées.

Nous allons à présent examiner les politiques d'ordonnancement avec priorités les plus répandues dans les bibliothèques de processus légers.

3.4.1 Priorités et exécution non-préemptible

Lorsque l'ordonnanceur fonctionne en mode non-préemptif, la « remise en cause » du processus léger actif ne peut s'effectuer que sur un point de synchronisation (cf. section 3.3.2.1). À ce moment, un ordonnanceur peut décider d'effectuer une commutation depuis le processus actif vers un processus de plus haute priorité. Par contre, puisque le processus actif ne peut pas être interrompu de manière asynchrone, il n'est pas possible de faire respecter à chaque processus un pourcentage d'occupation du processeur, par exemple.

En fait, d'une manière générale, il est seulement possible de veiller à ce que le processus actif soit toujours le processus prêt de plus haute priorité. Dans ce mode de fonctionnement, les priorités servent donc uniquement à établir une relation d'ordre entre les processus d'une application, mais ne peuvent en aucun cas être utilisées pour attribuer aux processus des « *vitesses d'exécution relatives* » ou autre. Concrètement, cela signifie que pour une application contenant deux processus légers p_1 et p_2, en considérant que p_1 possède une priorité de 1, fixer une priorité de 3 à p_2 aura le même effet que lui fixer une priorité de 100.

L'implantation d'un tel ordonnanceur est généralement une variante de l'ordonnancement

FIFO, dans laquelle la file des processus prêts est remplacée par une liste de processus ordonnée par ordre décroissant des priorités. Le processus actif est toujours le processus en tête de file. Les différences avec l'ordonnancement strictement FIFO sont les suivantes :

- Lorsque le processus actif exécute une instruction de commutation explicite, celle-ci n'est effectuée que s'il existe au moins un autre processus prêt de même priorité[9] que lui. Dans ce cas, c'est avec ce processus qu'est effectué le changement de contexte et le processus précédemment actif est inséré dans la liste conformément à sa priorité.

- Toutes les primitives pouvant déclencher l'éligibilité d'un nouveau processus ou le changement de priorité d'un processus prêt sont considérées comme des points de synchronisation, si bien que l'ordonnanceur teste s'il est oui ou non nécessaire d'effectuer une commutation de processus en chacun de ces points. La réponse est oui lorsqu'il existe un processus prêt de priorité supérieure à celle du processus actif.

- Les mécanismes de synchronisation tels que les verrous ou les sémaphores [55] ont également un comportement favorisant les processus de plus haute priorité. Par exemple, le « réveil » d'un sémaphore débloque non pas le processus bloqué depuis le plus longtemps en attente de ce sémaphore, mais le processus de plus haute priorité parmi les processus légers candidats.

Il découle des deux premiers points qu'à tout instant, la priorité du processus actif est supérieure ou égale aux priorités des autres processus prêts.

Bien que simple et efficace à mettre en œuvre, cette politique d'ordonnancement reste assez peu utilisée, principalement à cause de son manque de souplesse. En effet, il n'est pas rare de trouver des applications nécessitant des garanties un peu plus fortes sur la qualité de l'ordonnancement. Par exemple, s'il est souvent acceptable que l'ordonnanceur favorise sans nuance les processus de haute priorité par rapport aux processus de basse priorité, il semble légitime d'exiger qu'il traite avec un même égard les processus de priorité semblable. De par la nature d'un ordonnancement non-préemptif, cette dernière propriété ne peut pas être satisfaite dans ce cas.

3.4.2 Priorités et exécution préemptible

Les ordonnanceurs préemptifs offrent beaucoup plus de possibilités que leurs homologues non-préemptifs en matière de gestion des priorités. En effet, compte tenu de la possibilité qu'ils ont de réquisitionner le processeur à tout moment, les politiques de partage de temps les plus diverses peuvent être mises en œuvre.

Les deux politiques de partage de temps en fonction des priorités les plus répandues sont l'*ordonnancement temps réel* et l'*ordonnancement en tourniquet multiple*.

3.4.2.1 Échéancier temps réel

L'ordonnancement *temps réel* est une politique d'ordonnancement destinée aux applications possédant des exigences temporelles très strictes. Souvent, dans de telles applications, des **dates de fin au plus tard** sont attachées aux processus et c'est au système (donc à

9. Nous verrons dans le point suivant qu'il ne peut pas y avoir de processus prêts ayant une priorité plus grande que celle du processus actif. C'est un invariant de l'algorithme.

l'ordonnanceur) de faire en sorte que toutes les contraintes soient respectées, par exemple en ajustant dynamiquement les priorités des processus légers. La problématique est donc différente d'un ordonnancement avec priorités classique, où le programmeur définit lui-même les priorités mais ne pose pas de contrainte supplémentaire.

L'implantation d'un ordonnanceur temps réel **simple** peut s'articuler autour d'une unique liste de processus prêts. Pour chaque processus, l'ordonnanceur calcule le temps restant avant son échéance, ce qui représente l'inverse de sa priorité (plus l'échéance est proche, plus le processus devient prioritaire). La liste est alors triée par ordre décroissant de priorité (comme dans l'ordonnancement FIFO) et c'est le premier processus de la liste qui est actif.

Bien évidemment, un ordonnanceur temps réel est souvent plus complexe que ce modèle de base. Cependant, compte tenu du fait que cette catégorie d'ordonnancement est dédiée à une classe d'applications très spécifique, elle n'est pas disponible en standard dans les bibliothèques de processus légers à vocation générale. De plus, les applications visées par PM^2 ne concernant nullement le domaine du temps réel, nous nous contentons simplement de mentionner l'existence de cette politique d'ordonnancement sans nous y attarder plus longtemps.

3.4.2.2 Tourniquet multiple

L'ordonnancement en tourniquet multiple est une extension du modèle du tourniquet classique permettant la prise en compte de plusieurs niveaux de priorités d'exécution.

Dans sa version de base, l'algorithme a pour objectif d'assurer:

1. que le processus actif soit toujours choisi parmi ceux possédant la plus haute priorité;
2. qu'il y ait un partage de temps équitable entre les processus de même priorité.

L'implantation d'un tel algorithme peut être réalisée simplement en utilisant une structure de donnée à deux niveaux (figure 3.11). Au premier niveau, l'ordonnanceur gère une table des priorités (indicée de 1 à `max_prio`) dont chaque case contient la liste des processus prêts possédant la priorité correspondante. Au deuxième niveau, c'est-à-dire au sein d'une liste de processus légers de même priorité, l'ordonnanceur gère les commutations de contexte à l'aide du modèle du tourniquet.

Le principe est le suivant. À tout moment, il n'y a qu'un seul tourniquet *actif* (*i.e* en train de tourner). Il correspond au **premier tourniquet non-vide** rencontré par l'ordonnanceur lors d'un balayage de la table des priorités en partant de la priorité la plus grande. Un tourniquet reste actif tant qu'il n'est pas vide et tant qu'il n'existe pas de tourniquet non-vide de plus haute priorité que la sienne. Si le tourniquet actif devient vide, alors l'ordonnanceur en recherche un autre en parcourant la table dans le sens des priorités décroissantes. Si un tourniquet non-vide de plus grande priorité apparaît, alors celui-ci devient le tourniquet actif.

La figure 3.11 illustre un ordonnanceur à tourniquet multiple gérant quatre niveaux de priorités. Dans le cas de figure présenté, les processus légers prêts ont soit la priorité 1, soit la priorité 3. C'est donc le tourniquet contenant les processus légers de priorité 3 qui est activé, entraînant un partage de temps exclusivement entre ces processus. Il faudra attendre qu'il n'y ait plus de processus prêt de priorité 4, 3 ou 2 pour que les processus de priorité 1 puissent accéder au processeur.

On le voit, l'ordonnancement à tourniquet multiple respecte la hiérarchie des priorités et permet un ordonnancement préemptif au sein d'une classe de priorité. Cependant, il peut

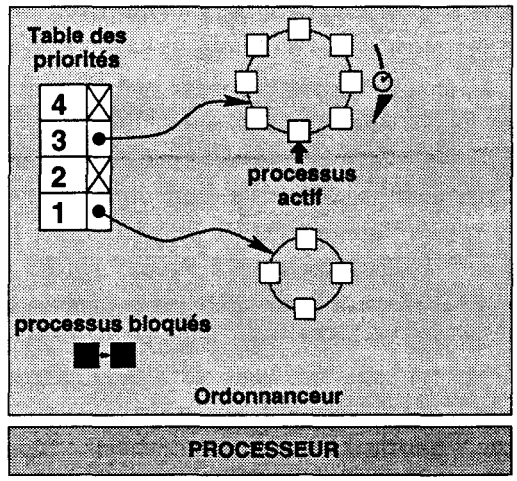

FIG. 3.11 - *Structure d'un ordonnanceur à tourniquet multiple.*

s'avérer gênant que le partage du temps processeur soit exclusivement effectué parmi les processus de plus haute priorité. En particulier, ce comportement peut facilement mener à des situations de *famine*, par exemple en présence d'un processus de haute priorité exécutant une boucle infinie.

C'est pour pallier à cet inconvénient que certaines variantes de cet algorithme ont été élaborées. Ces variantes consistent à diminuer régulièrement la priorité des processus appartenant au tourniquet actif, de façon à permettre à l'ordonnanceur d'exécuter périodiquement des processus prêts de plus faible priorité.

Un exemple de telle variante a été implantée au sein du noyau de processus légers MARCEL et sera décrite dans la section 6.3.3.

3.5 Contraintes techniques

Comme nous le verrons un peu plus loin (section 3.6 et 3.7), les nombreuses qualités des processus légers font de la multiprogrammation un outil bien adapté à la fois pour la construction de systèmes d'exploitation et pour le support des applications concurrentes.

Cependant, bien que bénéfique sur de nombreux plans, leur utilisation peut parfois s'avérer délicate, et cela pour plusieurs raisons.

3.5.1 Dimensionnement des piles d'exécution

Le dimensionnement des piles d'exécution allouées aux processus légers d'une application est probablement le problème le plus épineux rencontré par les programmeurs d'applications multiprogrammées [10]. En effet, la profondeur d'appel maximale que peut effectuer un processus léger dépend directement de la taille de sa pile d'exécution.

10. 90 % des bogues relevés dans les programmes des utilisateurs de PM^2 sont dus à l'utilisation d'une taille de pile insuffisante.

Plus cette taille est grande, plus le processus peut, par exemple, effectuer des appels récursifs profonds. Évidemment, cette constatation, effectuée de manière isolée, incite à l'utilisation systématique de très grandes tailles de pile pour les processus légers d'une application. En fait, cela soulève deux problèmes. Le premier est qu'une telle politique d'allocation des piles occuperait beaucoup trop de mémoire et ferait perdre du même coup un des intérêts principaux des processus légers : leur efficacité due à leur faible consommation de ressources. Le second est qu'il est très difficile, voire impossible, d'évaluer concrètement ce que représente une *très grande taille de pile*, d'une part parce que la profondeur d'appel d'un traitement n'est pas toujours prévisible *a priori*, et d'autre part parce que, pour un même traitement, la consommation de l'espace de pile d'un processus varie fortement suivant l'architecture du processeur sous-jacent et le compilateur utilisé.

Ces caractéristiques font que, la plupart du temps, le choix de la taille des piles d'exécution s'effectue par approximation grossière, ce qui augmente les risques de débordements. Cela nous amène directement à évoquer le deuxième problème inhérent aux piles des processus légers : le rattrapage d'erreur lors d'un débordement de pile. Dans beaucoup de bibliothèques de processus légers, il n'y en a aucun, c'est-à-dire que si un processus, pendant l'appel d'une fonction, sort de l'espace de pile qui lui est alloué, le système ne le détecte pas. Cela conduit souvent à l'écrasement de données d'autres segments de la mémoire et donc à la terminaison prématurée de l'application.

Pour remédier au premier problème, qui consiste uniquement à trouver le bon dimensionnement de la pile d'un processus léger, certaines bibliothèques fournissent un mécanisme d'extension dynamique des piles d'exécution. C'est par exemple le cas de la bibliothèque LWP (Sun [156]) ou encore de la bibliothèque MARCEL (LIFL). Le mécanisme fourni par LWP est réalisé grâce au concours du système d'exploitation (SunOs). L'allocation des piles initiales est faite de façon très espacée en mémoire virtuelle, ce qui permet, par la suite, de pouvoir utiliser l'espace intermédiaire pour leurs extensions. Les inconvénients de ce mécanisme résident dans le fait que, d'une part, ils sollicitent l'intervention du système d'exploitation (manque de portabilité) et que, d'autre part, la taille maximale des piles des processus est fixée de manière arbitraire. Le mécanisme fourni par MARCEL est décrit en section 6.3.4.

Pour tenter de remédier au second problème, qui consiste à détecter un débordement de pile (avant qu'il n'ait causé de dégâts) et à propager une erreur au niveau de l'application, certaines bibliothèques fournissent un mécanisme consistant à empêcher l'accès (en lecture et en écriture) d'une zone mémoire se situant dans le fond de la pile des processus. C'est par exemple le cas des bibliothèques Pthreads [126, 127] et LWP. Ainsi, lorsqu'un processus tente (malgré lui) d'utiliser cette zone, une exception matérielle est déclenchée et un signal est envoyé à l'application. Bien que fonctionnel dans la plupart des cas en pratique, ce mécanisme n'est pas fiable à 100 %, car il ne peut détecter un accès mémoire se situant encore au-dessous de la zone interdite. Pourtant, de tels accès peuvent se produire dans des programmes utilisant des fonctions comportant de larges structures de données locales non-initialisées.

3.5.2 Réentrance du code

Parmi tous les avantages conférés par l'utilisation des processus légers, celui d'autoriser un partage efficace de la mémoire est souvent mis en avant. Pourtant, par certains aspects, l'accès par un ensemble de processus légers à l'intégralité d'un même espace d'adressage pose des problèmes. En effet, s'il est intéressant pour un programmeur de pouvoir faire communiquer des processus concurrents par le simple biais de variables globales, il est, en revanche, très gênant

de constater que **toutes** les variables globales, y compris celles gérées par des bibliothèques dont le source n'est pas disponible, peuvent être accédées (directement ou indirectement) de la même manière.

Le problème se pose précisément avec les données qui présentent le risque d'être accédées, par un processus léger, en écriture concurremment avec d'autres accès en lecture ou en écriture. Lorsque ces données sont définies par le programmeur, la situation n'est pas dramatique car il est toujours possible de mettre en place une synchronisation régissant l'accès à ces données. En revanche, le problème est entier lorsqu'il s'agit de données gérées par des modules de bibliothèques dont il n'est pas question (ou tout simplement possible) de modifier le code source.

Une illustration typique de cette situation est fournie par le module bien connu de la bibliothèque C standard prenant en charge la gestion des allocations mémoire dynamiques (d'interface `<malloc.h>`). La majorité des primitives fournies par ce module modifient une structure de donnée globale représentant la liste chaînée des emplacements mémoire libres dans l'espace d'adressage du processus lourd. Par conséquent, l'exécution concurrente de certaines de ces primitives par plusieurs processus légers risque d'endommager la structure de donnée globale et d'aboutir rapidement à la terminaison prématurée du processus. Notons que, avec un ordonnancement non-préemptif, le problème ne se pose pas puisque cette situation ne peut pas survenir. En revanche, elle est très probable avec un ordonnancement préemptif. Dans ce cas précis, une solution souvent [11] applicable est de garantir l'accès en exclusion mutuelle à ce module, en entourant l'appel de chacune des primitives par un code de synchronisation.

Il y a malheureusement des cas de figure posant des problèmes auxquels même un ordonnancement non-préemptif n'échappe pas. Ces cas de figure concernent les primitives manipulant des données globales **et** contenant un ou plusieurs appels potentiellement bloquants. L'exemple parfait est la primitive de réception bloquante de messages fournie par la bibliothèque PVM nommée `pvm_recv`. Cette primitive modifie, lors de son appel, un certain nombre de données internes à PVM, qui sont par ailleurs également utilisées par la primitive `pvm_send` d'émission de messages. Il n'est donc pas question d'autoriser l'exécution simultanée (concurrente par plusieurs processus légers) de ces deux primitives. Cependant, il n'est pas possible, comme dans l'exemple précédent, de protéger ces appels pour qu'ils s'effectuent en exclusion mutuelle. En effet, l'appel de `pvm_recv`, potentiellement bloquant, risquerait alors de bloquer le processus appelant en pleine section critique, empêchant ainsi les autres processus d'accéder à la primitive `pvm_send` pendant un temps indéterminé. Par conséquent, à moins de modifier le code source de PVM [12], ce problème n'est pas solvable. Notons pour terminer que ce problème se manifeste également avec un ordonnanceur non-préemptif. En effet, un appel à `pvm_recv` peut occasionner un changement de contexte dû au blocage du processus appelant au beau milieu de l'exécution de la primitive. À ce moment, des données internes à PVM peuvent être dans un état ne supportant pas l'exécution de la primitive `pvm_send`.

3.5.3 Protection mémoire

Les problèmes de protection mémoire entre processus légers sont évoqués en section 3.2.1. Ils se traduisent par l'existence d'un risque de corruption des données du processus lourd

11. Mais pas toujours ! En particulier, il est très difficile de synchroniser les appels indirects, surtout s'ils sont eux-mêmes issus de la bibliothèque standard.

12. Ce que nous avons d'ailleurs effectué dans le cadre d'une implantation efficace de PM^2, voir chapitre *Implantation*.

provoqué par l'accessibilité de tout l'espace mémoire du processus lourd par n'importe quel processus léger. Par conséquent, un processus léger exécutant une suite d'instructions contenant un bogue peut, par un concours de circonstances malheureuses (*i.e.* par effets de bord), endommager le reste de l'application.

Ces problèmes sont malheureusement inhérents à l'implantation des processus légers en espace utilisateur et n'ont donc pas de solution satisfaisante dans ce cadre.

3.6 Utilisations du multithreading

Dans cette section, nous effectuons un tour d'horizon des divers domaines actuels d'utilisation [13] des processus légers. Il est intéressant de remarquer que les processus légers ne sont pas seulement utilisés pour des raisons de gain de performances, mais qu'ils sont également utilisés pour le modèle d'expression du parallélisme qu'ils fournissent et pour leur adéquation générale au support des applications comportant naturellement des composantes concurrentes à grain fin.

3.6.1 Les processus légers dans les systèmes d'exploitation

L'utilisation la plus massive des processus légers se situe sans nul doute au sein des systèmes d'exploitation (Solaris [141], Mach [68], Amoeba [129], etc.). Dans ce contexte, si les finalités varient beaucoup d'un système à l'autre, l'objectif premier reste bien souvent l'amélioration des performances.

3.6.1.1 Appels système

Certains systèmes d'exploitation utilisent les processus légers de manière interne au noyau pour prendre en charge l'exécution des appels systèmes, c'est-à-dire de traitements devant être exécutés par le noyau pour le compte d'applications s'exécutant normalement en espace utilisateur.

Le système TAOS (stations DEC Firefly [159]) en est une bonne illustration. Dans ce système, les processus légers remplacent les processus lourds traditionnels pour l'implantation de nombreux services. Lorsqu'un processus utilisateur effectue un appel de service, il déclenche un RPC local dont l'exécution est prise en charge par un processus léger « serveur ». Bershad et *al.* [10] ont proposé une optimisation de ce mécanisme, appelé LRPC (*Lightweight Remote Procedure Call*).

L'objectif des LRPC est de diminuer le temps nécessaire à la prise en charge de ces appels. Le principe de fonctionnement est le suivant. Lorsqu'un appel système doit être effectué, le noyau commence par préparer certaines zones de données, dont une pile d'exécution (*i.e.* un processus léger) pour prendre l'appel en charge. L'application peut ensuite placer les arguments de l'appel dans une zone spéciale accessible à la fois par l'application et le noyau. Une fois les arguments empilés, un déroutement donne le contrôle au noyau du système, qui modifie alors complètement la « cartographie mémoire » du processus utilisateur, de telle manière à ce qu'il puisse accéder aux données du serveur, tout en s'exécutant sur la nouvelle pile. L'exécution du service peut alors être effectuée par le processus utilisateur lui-même.

13. Ce tour d'horizon ne prétend pas être exhaustif, il a simplement pour objectif de mentionner les utilisations des processus légers les plus significatives.

Grâce à cette optimisation, qui utilise des fonctionnalités propres aux processus légers, les auteurs rapportent une efficacité remarquable, n'introduisant qu'un très faible surcoût comparé au minimum théoriquement réalisable.

3.6.1.2 Serveurs multiprogrammés

Dans certains systèmes d'exploitation (par exemple Amoeba [129]), les processus légers sont utilisés pour l'implantation de processus serveurs multiprogrammés. Les serveurs de fichiers sont particulièrement concernés par cette tendance. L'objectif est de diminuer les temps de réponses des accès en lecture en autorisant le traitement de plusieurs requêtes de façon concurrente.

Lorsque plusieurs requêtes sont adressées simultanément à un serveur de fichiers traditionnel, celles-ci sont prises en charge séquentiellement. Ce qui signifie qu'un des clients ayant émis une requête devra attendre pendant un temps égal à la somme des temps de traitement de chacune des requêtes. Pourtant, la plupart des dispositifs de mémoire secondaire acceptent de traiter plusieurs requêtes en parallèle. De plus, pendant la majeure partie du traitement d'une requête de lecture, le serveur de fichier est bloqué en attente des données en provenance du dispositif de stockage.

L'idée est donc de mettre à profit ces périodes d'inactivité du serveur pour lui faire traiter d'autres requêtes. En d'autres termes, il s'agit de recouvrir les délais d'accès disque par des calculs. Pour ce faire, les processus légers représentent l'outil le plus souple et le plus simple à utiliser. En effet, plutôt que de gérer une table des requêtes en cours de traitement et des changements de contexte de façon « manuelle », il est bien plus simple et plus sûr (et tout aussi efficace) de laisser faire ce travail par un ordonnanceur de processus légers. Dans ce cas, la programmation du serveur devient très simple, puisqu'il suffit de créer un nouveau processus léger à chaque fois que l'on désire prendre en charge l'exécution d'une nouvelle requête (figure 3.12).

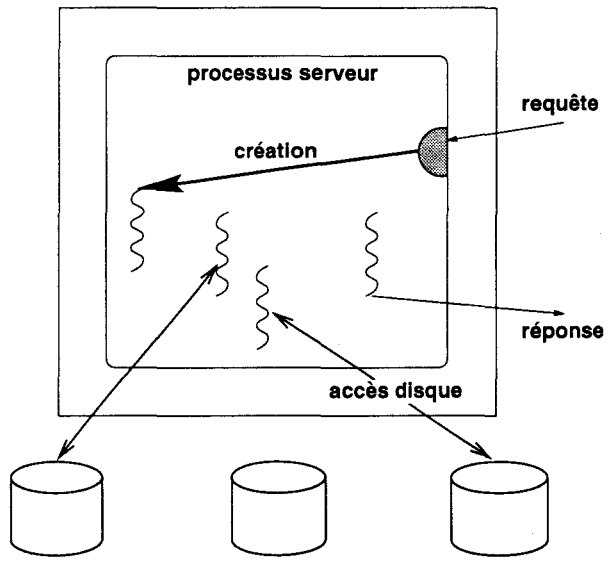

FIG. 3.12 - *Structure d'un serveur de fichier multiprogrammé. Chaque requête est prise en charge par un processus léger.*

3.6.2 Les processus légers dans les supports d'exécution des langages

3.6.2.1 Cible de compilateurs

La difficulté de concevoir des applications parallèles complexes directement au-dessus d'outils système de bas niveau a provoqué l'apparition d'un certain nombre de langages parallèles (Ada [160], CC++ [28], Opus [116], HPF [94, 95], PloSys [125], etc.). L'objectif est de fournir au programmeur des abstractions de haut niveau permettant de manipuler de manière aisée plusieurs entités actives au sein d'une même application. Si cette approche facilite souvent grandement la tâche du programmeur, elle complexifie beaucoup, en contrepartie, la réalisation du compilateur qui doit ainsi prendre en charge toutes les opérations relatives à la gestion du parallélisme (création d'une tâche parallèle, commutations, ordonnancement, synchronisations, communications, etc.).

Pour faciliter le développement de tels compilateurs et pour augmenter leur portabilité, certains concepteurs ont choisi des bibliothèques de processus légers pour cible de leurs générateurs de code. Ainsi, les exécutables générés par ces compilateurs n'utilisent pas directement les fonctionnalités du système, mais exploitent au contraire les fonctionnalités offertes par les primitives associées à la gestion de processus légers.

Les avantages d'une telle approche sont multiples :

- La complexité du compilateur est fortement atténuée, puisque celui-ci peut s'appuyer sur des primitives élaborées de gestion d'activités parallèles. Son développement est ainsi plus rapide.

- La portabilité du langage est augmentée, car les bibliothèques de processus légers sont souvent disponibles sur plusieurs architectures. De plus, si le compilateur n'utilise que les fonctionnalités « standard » des processus légers, son portage sur une nouvelle bibliothèque de processus légers ne nécessite qu'une adaptation superficielle.

- La fiabilité du compilateur est renforcée. D'une part, sa conception modulaire (gestion séparée du parallélisme) réduit les risques d'erreur. D'autre part, les bibliothèques de processus légers sont souvent intensivement testées par ailleurs (l'utilisation par des compilateurs n'étant pas leur seule finalité), ce qui augmente leur robustesse.

L'exemple le plus représentatif est fourni par le compilateur GNU pour le langage *Ada95* [128] qui génère du code utilisant la bibliothèque de processus légers *Pthreads* développée par Frank Mueller en Floride [126, 127].

Des approches similaires sont également suivies par des compilateurs de langages parallèles **distribués**. Dans ce cas, le code généré s'appuie non pas sur de simples bibliothèques de processus légers, mais sur des environnements distribués à base de processus légers. C'est le cas par exemple du langage *CC++* qui s'appuie sur *Nexus*, ou encore du langage *Opus* qui s'appuie sur *Chant*. Ces derniers environnements, proches de PM^2 par certains aspects, seront présentés en détail dans le chapitre 4 (*Parallélisme à grain fin et distribution*).

3.6.2.2 Supports d'exécution pour les langages à objets

Parmi les nombreux domaines où la programmation objet apporte les grandes qualités qu'on lui connaît, la modélisation des programmes parallèles ou encore la simulation du monde réel occupent une place privilégiée. C'est ainsi que, ces vingt dernières années, nous avons

assisté à la naissance d'un très grand nombre de réalisations conciliant objets et parallélisme (ou du moins concurrence). Ces réalisations comprennent des systèmes d'exploitation (SOS [150], Cool [108], Commando [113]), des langages (Box [81], Guide [105], Orca [5]) ou encore des bibliothèques de composants logiciels accompagnées d'un support d'exécution (projet Sloop [8, 7], TOSCA [83]). Beaucoup utilisent les processus légers pour supporter les multiples activités concurrentes et/ou parallèles introduites dans le modèle de programmation.

Les processus légers sont particulièrement bien adaptés au support de ces environnements pour plusieurs raisons :

Invocation à distance Dans un système à base d'objets distribués, les objets peuvent référencer d'autres objets situés sur des sites distants. Dès lors, un mécanisme d'invocation de méthodes à distance doit être supporté par le système. Ces invocations, qui s'apparentent à des appels de procédures à distance, sont très performantes lorsqu'elles sont implantées à l'aide de processus légers. En effet, le coût d'une création de processus léger est faible et, comme celle-ci est effectuée au sein du processus contenant l'objet, l'accès aux attributs de l'objet s'effectue de manière directe, sans nécessiter la moindre recopie. De plus, si la sémantique du langage le permet, il est possible d'effectuer plusieurs invocations de méthodes en parallèle sur un même objet sans nécessiter de mécanisme supplémentaire.

Invocation intra-domaine Beaucoup de langages utilisent le concept d'*objet actif* pour encapsuler la notion de flot d'exécution dans une entité « objet » [81]. Un objet actif est un objet classique possédant une activité interne exécutant une méthode particulière. Cette activité peut accéder aux attributs de l'objet (qui peuvent être des références sur d'autres objets) et invoquer des méthodes sur ces attributs. De ce fait, le flot d'exécution d'un objet actif peut, au gré des appels de méthodes, « traverser » un certain nombre d'objets du système. L'utilisation d'un processus léger pour supporter l'activité d'un objet actif s'avère très avantageuse dans ce cas, car il peut directement « traverser » les objets se trouvant dans le même processus lourd que lui. Au contraire, une implantation à l'aide de processus lourds mènerait à exécuter chaque appel de méthode non-local à un objet sous forme d'un appel de procédure à distance.

Parallélisme massif Les environnements à objets destinés au support d'applications de simulation du monde réel utilisent fréquemment la notion d'*acteur* [43, 1] comme entité active de base pour la construction des applications. Un acteur est une entité autonome communiquant avec les autres acteurs par envoi de messages. Pour faciliter la conception de ces systèmes, il est commode d'implanter les acteurs par des processus. La caractéristique principale de ces applications de simulation est qu'elles peuvent comporter des milliers d'acteurs interagissant simultanément. Dans ce contexte, les processus légers s'avèrent être des composants privilégiés pour le support de simulations à si grande échelle, puisqu'ils sont les seuls à pouvoir supporter l'expression d'une concurrence aussi massive.

3.6.3 Les processus légers dans les applications

De plus en plus d'applications utilisent les processus légers de manière directe pour exprimer efficacement et facilement le parallélisme naturel qu'elles contiennent.

Les applications comportant une interface graphique, par exemple, gagnent souvent à utiliser le concept de processus léger pour augmenter leur réactivité [3]. Le principe consiste à attacher un ou plusieurs processus légers à chaque élément de l'interface (bouton, menu, etc.). Ces processus, créés à chaque occurrence d'un événement, permettent de prendre en charge de façon *asynchrone* les traitements associés à cet événement. Cela permet, entre autres, de dégager le gestionnaire d'interface de ce travail pour le laisser se focaliser sur la détection des événements externes (souris, clavier). Ainsi, lorsqu'un utilisateur appuie sur un bouton de l'interface, un processus léger est créé pour prendre en charge le traitement associé à ce bouton. Et même si ce traitement est long, l'interface est immédiatement disponible pour accepter d'autres actions. La **réactivité** de l'application est de ce fait considérablement augmentée.

Un autre domaine d'application très concerné par l'utilisation des processus légers est celui des simulations. Dans ces applications, il s'agit d'observer le comportement collectif d'un certain nombre d'activités complexes souvent asynchrones. Les applications simulant le trafic automobile d'une grande ville, le fonctionnement d'un supermarché aux heures de pointe, ou encore la gestion des paniers et des cabines dans une piscine municipale [107] sont des échantillons représentatifs de cette catégorie d'applications. Dans ce contexte, l'utilisation de processus légers (1 processus léger = 1 activité de la simulation) simplifie énormément la programmation de la simulation. En effet, il est bien plus naturel de décrire le problème en spécifiant les *comportements* de chacune des entités actives de la simulation, que d'écrire un ordonnanceur spécialisé chargé de simuler l'évolution parallèle d'entités codées de manière synthétique dans des structures de données passives.

Pour les mêmes raisons, certains concepteurs de logiciels de jeux d'Arcades bien connus utilisent les processus légers lorsqu'il s'agit de faire évoluer plusieurs entités actives (monstres et autres robots) en parallèle. Des compilateurs C++ commerciaux spécialisés pour le support de ces « applications » intègrent d'ailleurs la notion de processus léger au sein de leurs bibliothèques.

3.7 Conclusion

Tout au long de ce chapitre, nous avons examiné le concept de processus léger, ses avantages et inconvénients, et ses domaines d'utilisation typiques. En particulier, nous avons observé que, par l'étendue de ses qualités, ce concept trouvait son utilité dans de nombreux domaines d'application.

Cependant, l'utilisation des processus légers pour le support d'applications de calcul « intensif », c'est-à-dire pour des applications dont les seules qualités exigées concernent les performances globales, n'est pas encore entrée dans les mœurs.

Il n'y a pas si longtemps, Andrew Tanenbaum écrivait à propos des processus légers : « *Personne ne proposerait sérieusement de réaliser un programme qui joue aux échecs en utilisant les processus légers car on n'en tirerait aucun avantage* » [158]. Un des objectifs de notre argumentation est de montrer qu'il a tort en affirmant cela.

De même, la « reconnaissance » des bonnes qualités des processus légers est le plus souvent limitée à un contexte local (par opposition à distribué). Il y a deux raisons à cela. La première est que le concept de processus léger est souvent associé à une notion de partage de mémoire efficace. Par conséquent, son utilisation sur architectures distribuées semble quelque peu contre nature. La seconde raison est que l'intégration de ce concept à un environnement distribué

pose de nouveaux problèmes sémantiques, qui ne trouvent pas de solution universelle. Par exemple, faut-il considérer les processus légers comme de simples processus classiques très efficaces, ou faut-il au contraire exploiter leurs caractéristiques spécifiques? Ces aspects, sur lesquels nous reviendrons au début du chapitre 4, sont encore peu expérimentés et entrent également dans le cadre de notre étude.

3.7.1 Processus légers et calcul parallèle distribué

À ce stade du document, nous pouvons déjà dégager un certain nombre de propriétés dont pourraient bénéficier les applications parallèles distribuées en utilisant des processus légers.

3.7.1.1 Qualités « intrinsèques » aux processus légers

Il y a d'abord les propriétés que l'on peut qualifier d'« intrinsèques » aux processus légers, car elles sont une conséquence directe de leurs caractéristiques de base.

Recouvrements des temps de communication Nous l'avons vu en section 3.6.1.2, l'utilisation des processus légers dans l'implantation d'un serveur de fichiers permet une meilleure utilisation des ressources de la machine. Le gain provient du recouvrement des délais d'accès aux disques par le traitement d'autres requêtes.

Dans un environnement distribué, les différents fragments (processus) d'une application parallèle doivent fréquemment communiquer entre eux pour échanger de l'information. Ces communications, en l'absence de mémoire commune, passent fréquemment par un dispositif d'acheminement de messages (réseau de communication) entre les différentes machines. Étant donné que les différents processus d'une application peuvent difficilement s'exécuter en mode totalement synchronisé, lorsqu'un processus de l'application attend un message en provenance du réseau, le message peut ne pas être disponible immédiatement. Dans ce cas, le processus est bloqué par le système et le processeur devient inutilisé si aucun autre processus n'est prêt à s'exécuter localement.

Nous avons vu dans la section 2.3.1 qu'il était généralement préférable d'éviter le placement de plusieurs processus lourds sur le même processeur, principalement pour des raisons d'efficacité.

Une bonne solution à ce problème est d'utiliser plusieurs processus légers concurrents au sein de chaque processus lourd. En procédant de la sorte, on diminue les risques d'apparition de périodes d'inactivité des processeurs : la plupart du temps, lorsqu'un processus léger se met en attente d'un message, il existe un autre processus léger prêt qui peut immédiatement occuper le processeur.

Extensibilité En raison de la capacité du multithreading à gérer plusieurs centaines de flots d'exécution concurrents au sein d'un processus lourd, il est possible de concevoir des applications distribuées manipulant des dizaines de milliers d'activités concurrentes, sans pour autant les programmer différemment que dans le cas où il s'agirait de gérer quelques dizaines d'entités parallèles seulement.

Cette caractéristique est très importante, car elle permet la conception d'applications massivement parallèles capables, dans une certaine mesure, d'appréhender un degré de parallélisme variable.

3.7.1.2 Conception indépendante de la topologie de l'architecture

Pour des raisons liées au grain de parallélisme fourni par le modèle des processus légers, la conception d'applications basées sur ce modèle permet de s'abstraire de la topologie de l'architecture cible. En effet, compte tenu du nombre de processus légers qui peuvent coexister au sein d'une même application, la programmation d'une application ne doit plus raisonner en termes de processeurs disponibles, mais uniquement en termes de tâches à exécuter en parallèle.

Bien évidemment, cela suppose un support d'exécution capable de répartir, voire même de réguler un ensemble de processus légers sur une architecture distribuée. Nous verrons, dans le chapitre 5 consacré à PM^2, la proposition que nous faisons dans ce sens.

Cette approche, qui permet d'éliminer le fossé trop souvent constaté entre la conception d'une application parallèle et sa mise en œuvre, possède en outre les deux avantages suivants par rapport à une approche classique :

Virtualisation Les processus légers peuvent être considérés comme des processeurs virtuels capables de prendre en charge les tâches des applications parallèles. Ainsi, le programmeur dispose d'une machine virtuelle contenant plus de 10 000 processeurs. Il peut donc focaliser son travail de conception sur le découpage logique de ses applications, en générant de nouvelles tâches à chaque fois que l'algorithme le permet.

Cette virtualisation de l'architecture simplifie fortement les deux étapes de développement des applications, à savoir la conception (qui tend à devenir plus générale) et la réalisation (qui doit se rapprocher le plus possible de la spécification de l'algorithme parallèle).

C'est pourquoi nous pensons qu'il est particulièrement judicieux de confier le travail consistant à placer dynamiquement les tâches de l'application sur les processeurs physiques de l'architecture à un support d'exécution basé sur le concept de processus légers.

Interblocages Contrairement à un modèle à base d'un nombre fixé de processus serveurs, dans lequel il est difficile de gérer un nombre élevé de dépendances simultanées (*i.e.* traitement bloqué en attente de résultats provenant d'un autre traitement, voir section 2.3.1), un modèle basé sur les processus légers permet d'exprimer sans limitation artificielle les dépendances entre les traitements.

À chaque fois qu'il faut exécuter un nouveau traitement en parallèle, on lance un nouveau processus léger pour le prendre en charge (figure 3.13). Si ce traitement nécessite, à un point de son exécution, la disponibilité de données calculées par d'autres traitements, le processus léger sous-jacent se bloque simplement en attente de ces données et libère le processeur pour d'autres processus.

3.7.1.3 Parallélisme et priorités

Peu d'environnements de programmation parallèle permettent d'attacher des priorités aux différents processus d'une application. Pourtant, les performances de certaines applications dépendent fortement de la capacité du système à exécuter *d'abord* (ou simplement *plus vite*) des actions jugées prioritaires par rapport à d'autres.

Une application implantant un algorithme de type *Branch & Bound* est un bon exemple d'application dont les performances dépendent de l'ordre dans lequel l'espace des solutions

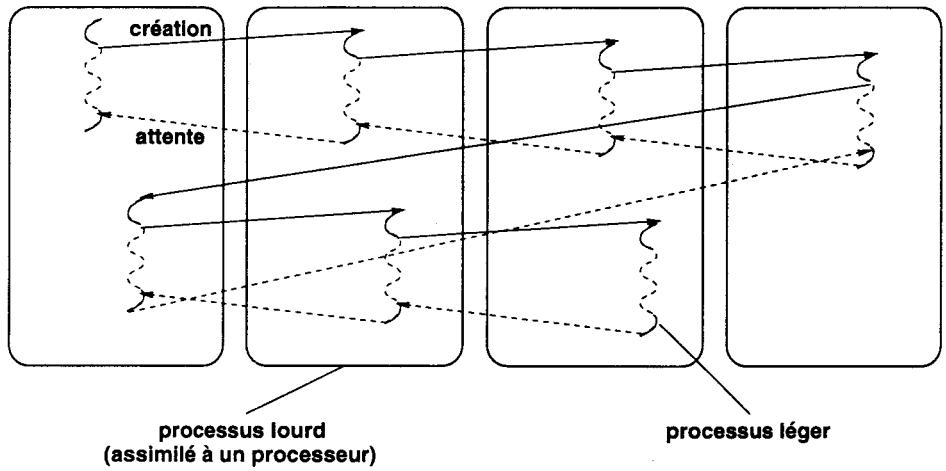

FIG. 3.13 - *La flexibilité des processus légers permet l'expression simple de chaînes de dépendance de taille supérieure au nombre de processeurs disponibles.*

est parcouru (cf. chapitre 2.1). Pour cela, des heuristiques sont fréquemment employées de manière à attacher des priorités d'exploration à chaque branche de l'arbre des solutions [58]. Dans le cadre de l'implantation parallèle d'une telle application, plusieurs branches de l'arbre sont explorées en parallèle. Avec une approche classique, il est souvent nécessaire de gérer une file (centralisée ou distribuée) de nœuds en attente d'exploration triés par priorité [46], ce qui complique la tâche du programmeur et peut constituer un goulot d'étranglement si le nombre de processeurs est élevé.

Au contraire, avec les processus légers, il est possible d'éviter le recours à la gestion d'une file de nœuds en attente de traitement, en utilisant simplement les propriétés de concurrence intrinsèques au modèle [54]. C'est dans ce contexte d'utilisation que la possibilité d'attacher des priorités aux processus revêt une utilité particulière. Nous reviendrons sur ce point dans le chapitre consacré à PM^2 (5).

3.7.1.4 Conclusion

Les quelques propriétés énumérées précédemment laissent entrevoir des possibilités intéressantes quant à l'utilisation des processus légers en contexte distribué pour le support des applications parallèles.

Cependant, l'intégration des processus légers dans un environnement distribué n'est pas du tout triviale, comme nous le verrons dans la première partie du chapitre suivant, et constitue souvent un compromis entre trois critères qui sont les *fonctionnalités*, la *portabilité* et l'*efficacité*.

Chapitre 4

Parallélisme à grain fin et distribution

La conclusion du chapitre précédent met en évidence un bon nombre de qualités relatives à l'utilisation des processus légers pour le support d'applications parallèles. C'est pourquoi le nombre de concepteurs d'applications les utilisant ne cesse de croître.[1] Pour la majorité des cas, cette utilisation s'effectue sur des machines multiprocesseurs à mémoire commune, pour lesquelles le modèle d'exécution des processus légers est, il est vrai, particulièrement adapté.

En revanche, les processus légers demeurent encore relativement peu utilisés en contexte distribué, et cela en dépit des qualités que nous avons évoquées précédemment, dont certaines sont pourtant spécifiquement liées à une utilisation dans ce cadre (par exemple le recouvrement des communications par du calcul). La principale raison expliquant ce phénomène réside dans la relative difficulté de concilier les deux concepts — *Multithreading* et *Distribution* — tant sur le plan du modèle de calcul que sur le plan de la mise en œuvre pratique.

Dans un premier temps, nous allons donc examiner dans quelle mesure une telle intégration est délicate et les principales manières de l'effectuer. Cela nous mènera ensuite à l'étude des principaux environnements de programmation parallèle réalisant actuellement une telle intégration. Enfin, nous terminerons par une synthèse sur ces environnements, en soulignant les insuffisances de chacun pour le support efficace des applications irrégulières massivement parallèles.

4.1 Comment intégrer processus légers et distribution?

À la base, le modèle des processus légers consiste en un ensemble de flots d'exécution concurrents situés dans un même espace d'adressage (cf. chapitre 3). Cette caractéristique fait de la *mémoire partagée* le vecteur naturel des communications inter-processus, et les outils tels que les sémaphores et moniteurs de Hoare représentent logiquement les seuls mécanismes de synchronisation disponibles dans l'interface standard des différentes bibliothèques de processus légers existantes.

De ce point de vue, il peut paraître contre-nature de vouloir plonger les processus légers dans un univers distribué, en risquant ainsi de les priver de leur fonctionnalité la plus « caractéristique » : l'interaction via une mémoire commune. En fait, la problématique ne se résume

1. cf. les nombreuses discussions dans le forum électronique `comp.programming.threads`.

pas à cette interrogation, car les « bonnes propriétés » des processus légers ne peuvent être réduites à leur capacité à exploiter la mémoire commune pour interagir. En particulier, ce sont surtout les propriétés relatives à leur faible consommation de ressources qui sont intéressantes pour le support des applications parallèles. L'intégration des processus légers dans un environnement distribué n'est donc pas une idée saugrenue, mais elle nécessite néanmoins la définition d'un nouveau « cadre d'utilisation » pour ces derniers.

D'une manière générale, il y a deux façons d'intégrer les processus légers dans un environnement distribué :

1. Le premier type d'intégration peut être qualifié d'approche *minimaliste*, car paradoxalement, il ne constitue pas réellement une « intégration » des processus dans un environnement distribué, mais tout au plus une « juxtaposition » des deux concepts.

2. Le deuxième type d'intégration consiste au contraire à fournir un modèle d'exécution centré sur la notion de processus léger et un modèle de programmation définissant les interactions entre ces derniers. Les processus légers constituent alors réellement la notion centrale de l'environnement.

Nous allons maintenant examiner plus précisément ces deux approches.

4.1.1 Approche minimaliste

Certains programmeurs utilisent de façon conjointe une bibliothèque de communication standard (PVM, MPI, etc.) et une bibliothèque de processus légers pour développer des applications distribuées. Plus précisément, ces applications se composent d'un certain nombre de processus lourds (par exemple des tâches[2] PVM) communiquant à l'aide des mécanismes fournis par la bibliothèque de communication. Ces processus lourds restent les seules entités visibles « d'un point de vue distant » et encapsulent des processus légers qui ne possèdent qu'une vision *locale* de l'application.

Clairement, une telle approche a pour objectif principal le recouvrement des communications par du calcul, en autorisant l'exécution de processus légers pendant que d'autres sont en attente de communication. Bien que cet objectif soit souvent atteint, une telle approche possède de nombreux d'inconvénients :

- Dans cette approche, l'utilisation des processus légers se limite quasiment à une « technique d'optimisation » permettant de diminuer le risque d'oisiveté des processus lourds. Véritablement « encapsulés » dans les processus lourds, ceux-ci n'apparaissent pas dans le modèle de programmation distribué de la plate-forme. Par conséquent, la plupart des avantages potentiels apportés par leur utilisation disparaissent (en particulier ceux liés à la notion de *processeur virtuel*) ou s'estompent fortement (tels que ceux liés à leur granularité).

- Étant donné l'absence d'une notion de *processus léger distribué*, la sémantique des primitives de communication reste liée aux processus lourds : les émetteurs et récepteurs des messages sont toujours des processus lourds. Cependant, dans l'optique d'un recouvrement des communications par du calcul, les processus légers doivent accéder à ces primitives de communication. Le programmeur est alors confronté à des problèmes

2. Voir description de PVM en section 6.2.

sémantiques épineux (*Qu'arrive-t-il lorsque deux processus légers exécutent la primitive de réception de message? Lequel obtient le message?*) qu'il doit souvent résoudre au prix d'une complication importante de son application.

- Comme nous l'avons déjà évoqué au chapitre précédent, l'utilisation de bibliothèques présentant une interface non-réentrante dans un contexte multiprogrammé pose des problèmes d'accès concurrents à des ressources critiques. En outre, ces bibliothèques ne sont classiquement pas prévues pour être utilisées par des processus légers[3] et ne présentent donc aucune pré-disposition à une éventuelle coopération avec un ordonnanceur de tels processus. Par conséquent, leur utilisation reste très délicate et s'effectue encore une fois au prix d'un effort de programmation sensible.

- Les deux points précédents montrent l'effort de programmation généralement exigé par une intégration des processus légers dans un environnement distribué. Une utilisation « brute » de ces mécanismes au sein des applications atténue fortement la réutilisabilité de l'approche, a fortiori si l'application a des besoins très spécifiques en terme de communications (ce qui lui ôte tout espoir de généricité). Par conséquent, cet effort de programmation doit souvent être réitéré lors du développement d'une nouvelle application distribuée.

Tous ces inconvénients mettent deux points en évidence : 1°) l'intégration des processus légers dans un environnement distribué, qui n'est pas une tâche aisée, constituerait une approche bien plus réutilisable si elle consistait en la conception d'un environnement distribué *multithreadé* générique ; 2°) le modèle de programmation proposé doit être centré sur la notion de processus léger et non sur celle de processus lourd, de manière à en répercuter les propriétés au travers d'une architecture au lieu de les limiter à un contexte purement « local ».

4.1.2 Approche intégrée

Étant donné les nombreux inconvénients de l'approche précédente, un certain nombre de recherches ont été — et sont encore — menées sur le thème de la conception et la réalisation d'environnements de programmation parallèle distribués basés sur les processus légers. Les environnements résultant de ces recherches proposent tous un modèle de programmation centré sur la notion de processus léger mais se distinguent par les fonctionnalités d'interaction entre processus qu'ils fournissent. Il existe plusieurs classes d'interaction, que l'on assimile généralement au « modèle de programmation » proposé par les environnements concernés. Voici quelques approches possibles parmi les plus usuelles :

Processus légers distants Cette approche propose d'étendre, de façon plus ou moins transparente, le cadre d'utilisation des primitives « classiques » de gestion de processus légers (`pthread_create`, `pthread_mutex_lock`, ...) à un cadre global à l'architecture. Dans l'idéal, une application distribuée utilisant les processus légers se programme donc de manière analogue à une application *multithreadée* traditionnelle. Toutefois, il convient de nuancer fortement cette caractéristique d'un point de vue pratique, car les performances d'une telle approche sont évidemment catastrophiques. C'est pourquoi seules quelques approches restreignant ce modèle à l'extension de quelques fonctionnalités ont

3. On dit qu'elles ne sont pas *thread-aware*.

effectivement été proposées. Des environnements tels que RThreads [59] ou, dans une moindre mesure, Chant [91] ont suivi cette voie.

Processus communicants Dans les environnements distribués à gros grain, l'approche « processus communicants » demeure la plus utilisée. Aussi est-il naturel qu'une grande proportion des environnements distribués à grain fin adopte également une telle approche. Dans ce cas, chaque processus léger peut potentiellement communiquer avec n'importe lequel de ses homologues, pourvu qu'il ait connaissance de son identifiant. Notons que dans la plupart des environnements de ce type, les communications sont asynchrones. PVC [148], TPVM [65], DTMS [35] ou encore Chant [91] en sont des exemples représentatifs.

Relations client/serveur Certaines approches proposent un modèle d'exécution fondé sur la prise en charge d'appels de procédures (potentiellement à distance) par des processus légers. Ce type d'interaction introduit des relations clients/serveurs entre les différents processus légers d'une application et s'avère être particulièrement adapté aux applications issues de la parallélisation d'un algorithme séquentiel. Les environnements Nexus [73], Athapascan-0a [30], DTS [19] ou encore PM^2 [131] sont très représentatifs de cette approche.

Les deux derniers modèles de programmation cités représentent les tendances les plus suivies par les environnements actuels.

4.2 Principaux environnements existants

Dans cette section, nous allons examiner quelques environnements « représentatifs » des tendances évoquées précédemment. Nous avons regroupé ces environnements par « modèle de programmation », aussi commencerons nous par aborder les environnements basés sur l'**envoi de messages** entre processus avant d'examiner ceux basés sur des **appels de procédure à distance**.

4.2.1 Environnements basés sur l'envoi de messages

Les environnements présentés dans cette section proposent tous un modèle de programmation principalement fondé sur un ensemble de processus légers communiquant par échange de messages.

4.2.1.1 PVC

Le projet PVC (Processeur Virtuel de Classe) a débuté à l'université de Lille en 1990 sous l'impulsion d'Éric Delattre et de Jean-Marc Geib [39]. L'objectif était de proposer un support d'exécution pour les langages à objets parallèles sur architectures distribuées. Ce projet a, entre autres, mené à la définition des Composants Actifs de Communication (CAC [38, 40]), qui sont une proposition de structuration des applications parallèles en entités actives simples.

Les principaux objectifs du support d'exécution des CAC étaient :

- une bonne adéquation au support des applications à objets parallèles ;
- une indépendance des fonctionnalités vis-à-vis du système d'exploitation ;

- une portabilité aisée sur différentes architectures (machines multiprocesseurs, réseau de stations de travail, ...) ;
- une transparence de l'architecture sous-jacente.

Bien que le support des applications massivement parallèles ne soit pas l'objectif premier du projet PVC, nous le présentons ici pour deux raisons. La première est que ce projet était le prédécesseur du projet ESPACE et la majeure partie des personnes qui travaillent au sein du projet ESPACE ont également participé au projet PVC. La deuxième est que toute une série d'expérimentations a été menée afin d'évaluer l'aptitude des CAC à supporter les applications massivement parallèles. En particulier, un algorithme de lancer de rayons et un simulateur parallèle pour un langage fonctionnel ont été développés avec l'environnement des CAC. Ces deux circonstances font que l'expérience acquise lors du projet PVC a eu de larges répercussions sur le modèle de programmation PM^2 et sur sa réalisation.

Modèle de programmation Le modèle d'exécution de l'environnement PVC consiste en un ensemble de CAC communiquant par envoi de messages. Le CAC représente la brique de base pour la construction d'applications parallèles. Il se compose d'une activité, d'une boîte aux lettres et d'un environnement privé (figure 4.1), ce qui lui confère une structure proche de celle d'un *Acteur* [1]. En fait, seule la boîte aux lettres d'un CAC est accessible de l'extérieur : elle constitue son *interface*. Lorsqu'un CAC est créé dans le système (forcément par un autre CAC), une référence sur sa boîte aux lettres est retournée au CAC créateur. Cette référence, qui est un nom global au système, permet d'identifier de façon unique et transparente le CAC ainsi créé. Il devient ensuite possible, pour tout possesseur de cette référence, d'envoyer des messages au CAC.

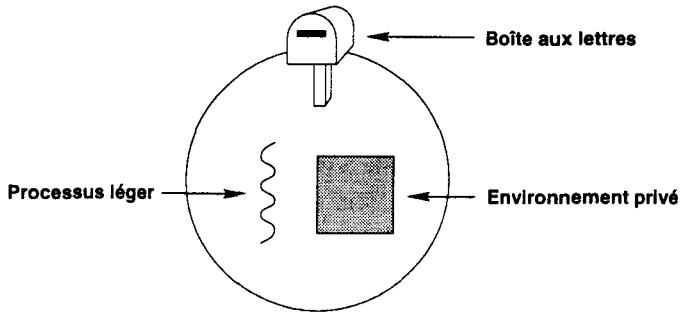

FIG. 4.1 - *La structure d'un Composant Actif de Communication.*

L'envoi de messages entre CAC s'effectue de manière asynchrone, toujours en mode point-à-point. Il consiste simplement à déposer un message dans la boîte aux lettres du CAC destinataire, qui est le réceptacle de tous les messages qui lui sont destinés. Les opérations d'extraction de messages de la boîte aux lettres doivent être effectuées explicitement par l'activité de ce dernier, car il n'existe pas de mécanisme d'interruption ou de déroutement qui pourrait avertir un CAC en cours de calcul qu'un nouveau message vient d'être déposé dans sa boîte aux lettres.

L'activité d'un CAC consiste en l'exécution d'une *fonction comportementale* par un processus léger. Cette fonction, décrite par le programmeur, doit donc gérer le comportement

interne du CAC (calculs et données locales) ainsi que ses communications avec les autres CAC. Le code des différentes fonctions comportementales d'une application doit être partitionné en terme d'unités logiques appelées des *modules*.

Un module est un contexte d'exécution pour les CAC. Il contient le code d'une partie des fonctions comportementales de l'application, un espace mémoire destiné à la création des CAC, ainsi qu'un CAC « *gestionnaire de module* » (figure 4.2). Ce dernier, dont la présence n'est pas visible par le programmeur, s'occupe principalement de la gestion des créations des CAC dans le module. En particulier, c'est lui qui reçoit les requêtes de création de CAC, les exécute et enfin renvoie les identificateurs de boîte aux lettres aux initiateurs des créations.

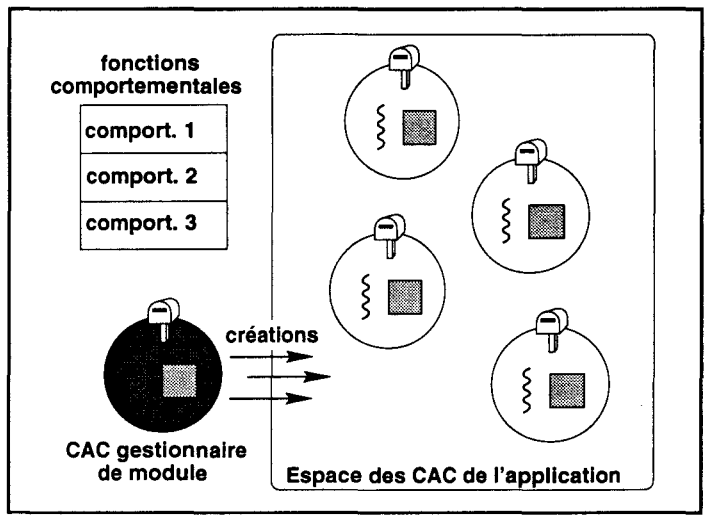

FIG. 4.2 - *Structure d'un module.*

Citons pour en terminer avec la description du modèle qu'une demande de création de CAC ne stipule pas où (*i.e.* dans quel module) la création doit être effectuée. Les seuls arguments sont le nom de la fonction comportementale à exécuter et les paramètres de lancement. À l'aide du nom de la fonction, le support d'exécution détermine quels sont les modules candidats à l'accueil du nouveau CAC et un mécanisme distribué de répartition de charge [96] s'occupe de choisir le meilleur[4].

Implantation Le support d'exécution des CAC a été porté sur une machine parallèle à base de transputers (MultiCluster II de Parsytec sous Helios), sur un réseau de stations de travail Sun (SunOs 4.1) et sur une ferme de processeurs ALPHA (OSF-1).

Sur le MultiCluster II, les modules sont implantés par des tâches Helios (processus lourds) et les CAC par des processus Helios (processus légers). Nous ne détaillerons pas davantage cette implantation car elle n'est plus utilisée aujourd'hui, la machine étant devenue obsolète.

Sur le réseau de stations Sun et sur la ferme de processeurs ALPHA, les modules sont implantés par des processus UNIX et les CAC par des processus légers. La bibliothèque LWP [156] a été utilisée sur les stations Sun et la bibliothèque DCE Threads, au standard POSIX, a été utilisée sur les processeurs ALPHA. Ces noyaux de multiprogrammation sont tous deux des bibliothèques livrées avec le système d'exploitation. Entre autres, cela permet aux processus

4. *i.e.* le moins chargé *a priori*.

légers de pouvoir utiliser des primitives d'Entrées/Sorties bloquantes de façon directe, sans risquer de bloquer le processus UNIX tout entier.

Le lancement d'une application sur la plate-forme du projet PVC est toujours précédé d'une phase de mise en place des modules sur les différents nœuds de l'architecture. Durant cette phase, des processus démons sont lancés et interconnectés entre eux par des liens de communication (maillage complet). Une fois ce maillage établi, les modules de l'application sont lancés sur les différents nœuds de l'architecture et leur exécution commence. Dès lors, grâce aux processus démons, chaque module possède un canal de communication avec tous les autres (figure 4.3).

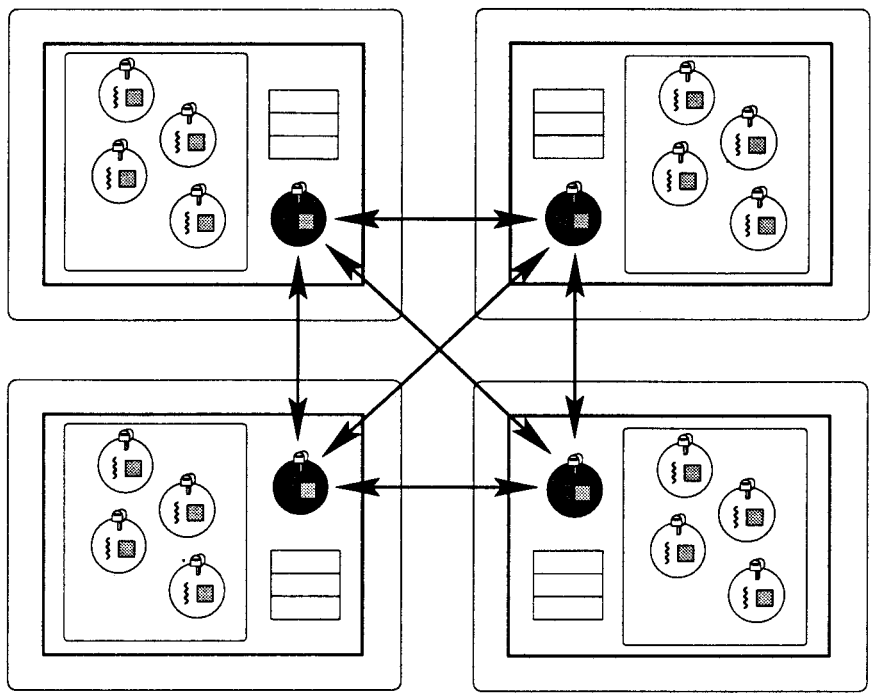

FIG. 4.3 - *Structure d'une application parallèle basée sur le support des CAC.*

Les communications entre CAC ont été implantées en utilisant directement les *sockets* TCP/IP en mode datagramme [41]. Le principe de fonctionnement est le suivant. Lorsqu'un CAC *a* désire envoyer un message à un CAC *b*, il émet ce message sur le canal de communication (*i.e.* la *socket* UNIX) en direction du module contenant *b*. Sur ce dernier module, le CAC *gestionnaire de module*, qui est en permanence à l'écoute des requêtes provenant de l'extérieur, extrait le message du canal et le dépose dans la boîte aux lettres du CAC *b*.

Le principe de fonctionnement d'une création de CAC est à peu près similaire, sauf qu'un deuxième message est émis vers le CAC initiateur pour lui communiquer la référence du nouveau CAC créé.

Grâce au fait que les implantations de l'environnement sont toutes basées sur des bibliothèques de multiprogrammation livrées avec le système d'exploitation et sur des fonctionnalités de communication de bas niveau elles aussi supportées par le système (*sockets*), il n'a pas été nécessaire de mettre en place des mécanismes de scrutation périodique du réseau, comme c'est le cas dans beaucoup d'autres environnements multiprogrammés (cf. Nexus, Chant ou

Athapascan).

Discussion L'objectif du projet PVC était de proposer un support d'exécution pour langages à objets parallèles. Les Composants Actifs de Communication sont une proposition de structuration des activités parallèles en entités simples communiquant par envoi de messages. Cette proposition a été validée par la conception et le développement d'un langage parallèle à objets utilisant la plate-forme des CAC comme support d'exécution [77].

Ce langage à objets, nommé BOX [81], fournit divers mécanismes permettant la gestion de la distribution et du parallélisme dans les applications à objets. En particulier, les flots d'exécution peuvent être manipulés par instantiation de la classe `FRAG`, qui désigne toute entité active dans le langage. Le compilateur du langage BOX génère directement du code CAC, ce qui permet aux applications BOX de tirer parti des divers outils disponibles sur la plate-forme des CAC. Ces outils comprennent un répartiteur de charge intégré [93] et un mécanisme de réexécution permettant la mise au point et le déverminage des applications distribuées [147, 146].

L'objectif du projet PVC ayant été atteint, une partie de l'équipe s'est alors intéressée au support des applications irrégulières massivement parallèles, qui allait d'ailleurs constituer le cœur du projet ESPACE par la suite. Dans ce contexte, il était naturel de mener les premières expérimentations à l'aide de la plate-forme des CAC. Comme indiqué en introduction, les premières expérimentations ont consisté en un algorithme de lancer de rayons et un simulateur parallèle pour un langage fonctionnel. Puis a suivi le développement d'une application d'optimisation combinatoire classique résolvant un problème d'affectation quadratique.

Pourtant, bien qu'assez facilement développées à l'aide de la plate-forme des CAC, ces applications n'ont pas exhibé les performances escomptées. La principale cause était qu'un surcoût important dû au coût des appels aux primitives de gestion mémoire était observé. La gestion des files de messages dans les boîtes aux lettres en était d'ailleurs la principale source. De même, la latence observée lors de la création de certaines activités était également trop importante. Ce dernier point était dû au fait que chaque création des CAC implique deux envois de messages (un aller et un retour) dans le système.

En fait, bien que le paradigme de l'envoi de message soit un modèle bien adapté au support des langages à objets, il s'avère qu'il l'est beaucoup moins au support de certaines applications parallèles. Ainsi, si le retour d'une référence de boîte aux lettres est utile dans des applications nécessitant une *coopération* entre les différentes activités, il est inutile dans le cas où l'on veut simplement déclencher un traitement à distance et récupérer un résultat *à la fin* de celui-ci. De même, la gestion de files de messages destinés à être traités séquentiellement par une activité ne semble pas être une fonctionnalité utile dans les applications massivement parallèles, où il vaut souvent mieux effectuer le traitement dès son arrivée sur le site. Nous verrons dans la section consacrée à l'étude de PM^2 qu'il est même possible de n'effectuer qu'une seule allocation mémoire pour stocker à la fois le contenu de la requête et la pile d'exécution de l'activité qui doit la traiter.

L'étude approfondie du comportement des applications massivement parallèles développées en utilisant des primitives d'envoi de message nous a fait comprendre l'influence des mécanismes de gestion mémoire sur les performances de l'exécution et l'importance d'utiliser un modèle d'exécution spécialisé pour le support des applications massivement parallèles.

4.2.1.2 Chant

Chant [88, 89] est un support système introduisant l'utilisation de processus légers communicants (« *Talking Threads* ») dans un environnement à mémoire distribuée. Il a été conçu à l'Institut d'Applications Informatiques en Sciences et Ingénierie du Centre de Recherche de la NASA (Hampton) par Matthew Haines, Piyush Mehrotra et David Cronk à partir de 1993.

La conception de Chant part d'un constat: les processus légers, malgré toute l'utilité que l'on s'accorde à leur reconnaître dans les systèmes à mémoire partagée, sont très peu exploités dans les environnements à mémoire distribuée. La principale cause est qu'il n'existe pas de standard actuel proposant un modèle à base de processus légers communicants pour architectures distribuées. C'est pourquoi Matthew Haines et son équipe proposent un système nommé Chant, fournissant un modèle capable de supporter des communications point-à-point entre les différents processus légers d'une application.

L'objectif visé à travers Chant est triple:

Portabilité Pour assurer une portabilité maximale au support d'exécution Chant, les concepteurs appuient son implantation sur des standards de multiprogrammation (POSIX Threads) et de communication (MPI) conçus eux-mêmes de façon à être portables.

Efficacité Tout en étant portable, le support d'exécution Chant doit s'efforcer de n'ajouter qu'un surcoût minime aux mécanismes de bases utilisés. Par exemple, aucune recopie mémoire « superflue » (ou disons plutôt « évitable ») ne doit être effectuée, de façon à garantir un surcoût *constant* par rapport au volume des données manipulées. De plus, chaque couche du support d'exécution Chant doit rester directement accessible par les applications, de manière à ce qu'elles puissent utiliser le couple *(fonctionnalités, efficacité)* qui leur convient le mieux.

Utilité De manière à ce que Chant soit effectivement utilisable par une large communauté de « parallélistes », plusieurs fonctionnalités ont été construites au-dessus du mécanisme d'envoi de message point-à-point de manière à fournir une large panoplie d'outils de programmation parallèle.

Actuellement, l'environnement Chant est principalement utilisé en tant que support d'exécution pour le code généré par des compilateurs de langages data-parallèles [116, 90].

Modèle de programmation Le modèle de programmation de Chant est avant tout un modèle basé sur des processus (légers) communicants. Bien que les différentes couches du support d'exécution fournissent des mécanismes comme les communications de groupe ou encore l'appel de services distants, le mécanisme de base est l'envoi de message point-à-point entre deux processus légers. Tous les autres mécanismes présents dans Chant sont bâtis sur celui-ci, qui a donc été conçu de façon à être le plus efficace possible.

Pour rendre possible la communication point-à-point entre processus légers dans un environnement distribué, il faut mettre en place un mécanisme de nommage global des processus dans le système (cf. nommage global des boîtes aux lettres dans l'environnement PVC). Dans ce but, les concepteurs de Chant ont fait le choix d'étendre l'interface POSIX-Threads en lui ajoutant une nouvelle catégorie d'entité, le processus *Chanteur*[5], permettant l'accès transparent aux processus légers d'une application.

5. *chanter thread* en anglais, mais le traducteur a visiblement des problèmes avec les faux-amis...

Les processus Chanteurs sont des processus légers dont les identifiants sont uniques (globaux) sur l'ensemble de l'architecture. Un tel identifiant est un couple formé de l'identifiant du processus léger qu'il désigne ainsi que de l'identifiant du contexte (processus UNIX) qui le contient. Un processus Chanteur est donc un processus léger que l'on peut manipuler à travers toute l'architecture distribuée (figure 4.4).

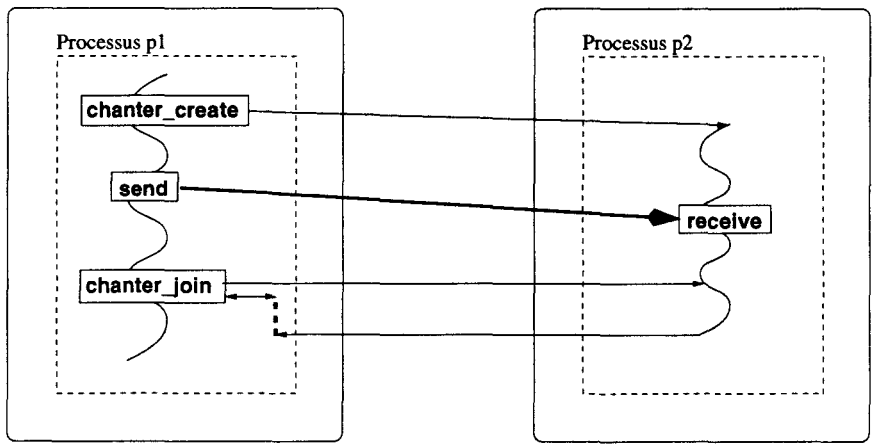

FIG. 4.4 - *Les processus Chanteurs peuvent être manipulés à distance de façon transparente. Les communications point-à-point s'effectuent via les primitives* `send` *et* `receive`.

La plupart des opérations usuelles disponibles sur les processus légers « locaux » (création, attente de terminaison, ...) ont été développées pour les processus Chanteurs. Par exemple, la primitive `pthread_create`, qui déclenche la création d'un processus léger local, s'appelle par analogie `pthread_chanter_create` lorsqu'il s'agit de déclencher la création d'un processus chanteur. Selon que la création du processus s'effectue dans le même contexte ou à distance, les mécanismes sollicités seront différents. Nous reviendrons sur ce point dans la section suivante. Pour des considérations d'efficacité, les primitives de synchronisations POSIX (moniteurs et conditions) n'ont pas été étendues de manière à pouvoir être utilisées à distance.

Les communications entre processus Chanteurs sont directement inspirées du standard MPI [121]. Un processus Chanteur peut envoyer un message à n'importe quel autre processus Chanteur pourvu qu'il possède son identifiant (figure 4.4). Les primitives de base se nomment *send* et *receive*, et conservent la sémantique des opérations d'envoi et de réception de messages conventionnelles. L'implantation efficace de ces primitives n'est pas triviale et sera également discutée dans la section suivante.

Bien que constituant la base de l'interface de Chant, le mécanisme d'envoi de messages entre processus Chanteurs ne convient pas à tous les besoins des applications parallèles distribuées. En particulier, des mécanismes de récolte d'information (production de traces, communication de la charge) ou de mise à jour de variables globales sont souvent nécessaires dans les applications parallèles. Par exemple, un processus s'exécutant dans un contexte (processus UNIX) a peut avoir besoin d'obtenir une copie d'une variable située dans un autre contexte b (accès mémoire distant). Il serait laborieux et peu naturel de réaliser cette opération à l'aide du mécanisme d'envoi de messages dans une application. En effet, pour envoyer un message de requête vers le contexte b, il faut posséder l'identifiant d'un processus Chanteur s'exécutant à l'intérieur qui soit prêt à rendre ce service, c'est-à-dire à lire le contenu de la variable et à

renvoyer sa valeur au processus appelant.

Ce dernier exemple est bien représentatif de la contrainte imposée par le mécanisme d'envoi de message : il faut toujours connaître son interlocuteur et il faut qu'il existe avant même l'établissement de la communication. Le mécanisme de *demande de service distant* ne possède pas ces inconvénients et se montre donc parfaitement adapté à la prise en charge d'actions telles que celle évoquée dans l'exemple. Il consiste en la possibilité de déclencher un traitement (*i.e.* l'exécution d'une fonction) dans un contexte distant.

Pour les raisons que nous venons d'évoquer, le mécanisme de *demande de service distant* a donc été intégré à Chant. Son fonctionnement général est le même que le RSR de Nexus, le RPC d'Athapascan ou encore le LRPC de PM^2 (se reporter à la section présentant ces environnements pour plus de détails).

L'environnement Chant étant principalement destiné au support des applications data-parallèles [90], il était souhaitable d'offrir des fonctionnalités permettant de manipuler des groupes de processus légers, de manière à pouvoir ensuite aisément exprimer un certain nombre d'*opérations collectives*. Le concept proposé par Chant pour supporter ces fonctionnalités s'appelle une *Corde*[6] [92] (figure 4.5).

Une Corde est une entité logique désignant un groupe de processus Chanteurs s'exécutant au sein d'une même application. Ces processus peuvent être localisés dans des contextes différents. Chaque processus possède un rang unique au sein de la Corde, ce qui permet, pour un processus, de pouvoir accéder à l'identifiant de ses voisins en utilisant un *index* logique, qui est un moyen d'expression du voisinage communément utilisé dans les algorithmes data-parallèles.

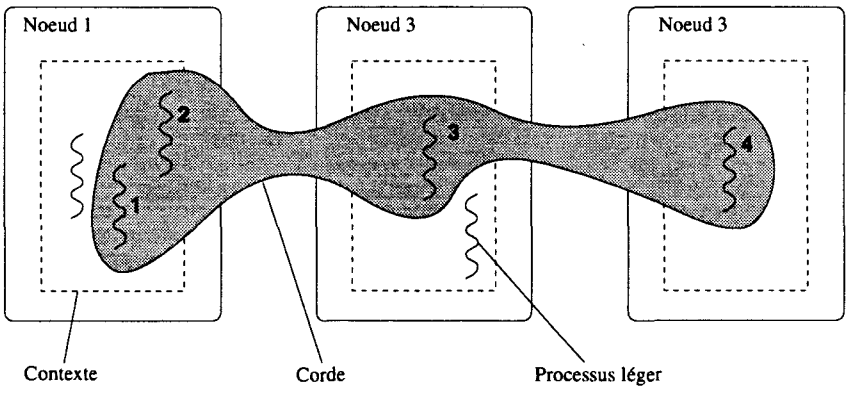

FIG. 4.5 - *Une Corde est un groupe de processus Chanteurs. Chaque processus possède un rang (index) unique au sein d'une Corde.*

À tout moment, un nouveau processus peut adhérer à une Corde, car cette dernière est une structure extensible. Cependant, les différents processus d'une Corde ne peuvent pas la quitter de façon isolée, car cela poserait des problèmes de renumérotation au sein de la corde qui seraient très délicats à résoudre. Hormis les opérations d'adhésion à une Corde, les fonctionnalités apportées par cette notion sont classifiables en deux catégories.

La première catégorie concerne les communications. Un processus appartenant à une corde peut envoyer des messages aux autres membres sans connaître leurs identifiants. Deux possi-

6. Une Corde est un ensemble de fils (threads) entrelacés...

bilités existent. Une première forme de communication permet un envoi de message point-à-point en spécifiant uniquement un rang dans la Corde en guise de destination. Ce mécanisme est surtout destiné aux applications formant des groupes de processus organisés en grilles ou en vecteurs. Dans ces applications, les processus ont besoin d'échanger des informations avec leurs voisins immédiats, qu'ils peuvent aisément désigner grâce au mécanisme d'indexation au sein d'un groupe. Une deuxième forme de communication permet la diffusion d'un message à tous les membres de la Corde.

La deuxième catégorie de fonctionnalités concerne la synchronisation. Elle n'en comprend qu'une seule pour l'instant: les barrières de synchronisation. Ce mécanisme permet à tous les processus d'une Corde de franchir de façon *synchrone* une étape de leur exécution. Lorsqu'un processus appelle la procédure de franchissement de barrière (`rope_barrier`), il reste bloqué jusqu'à ce que tous ses compagnons en aient fait autant. Une fois encore, l'utilité de ce mécanisme est importante dans les applications data-parallèles, car celles-ci nécessitent souvent de (re-)synchroniser les processus entre eux, par exemple avant une étape d'échange d'information.

Nous allons à présent décrire les grandes lignes de l'implantation du support d'exécution Chant.

Implantation Chant est actuellement opérationnel sur deux types d'architectures: les réseaux de stations de travail Sun et les machines Intel Paragon. Malgré le nombre de fonctionnalités supportées, il est très facilement portable car les ressources qu'il utilise (multiprogrammation et communications) sont restreintes et clairement définies par une interface (l'*interface système* de Chant) qui représente la seule portion de code à modifier lors du portage sur une nouvelle architecture.

La structure de Chant consiste en une superposition de plusieurs couches logicielles (figure 4.6). L'implantation de chaque couche s'appuie sur les fonctionnalités offertes par les couches inférieures.

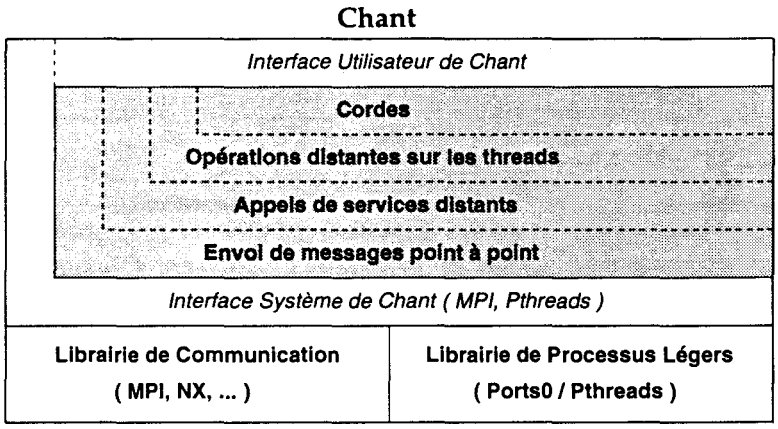

FIG. 4.6 - *Structure du support d'exécution Chant.*

La couche la plus basse du support d'exécution est l'*interface système* de Chant. Pour garantir une portabilité élevée, elle fournit aux couches supérieures une interface exclusivement constituée de primitives POSIX pthreads (pour ce qui concerne la multiprogrammation) et de primitives MPI (pour ce qui concerne les communications). C'est la seule couche qui

doit être adaptée à l'architecture sous-jacente et en particulier aux bibliothèques de multiprogrammation et de communication disponibles. Par exemple, sur la machine Intel Paragon, la bibliothèque propriétaire NX (Intel) a été utilisée, par conséquent les quelques fonctionnalités MPI de l'interface système de Chant ont été réécrites en utilisant les primitives NX. L'adaptation du noyau de multiprogrammation pour le système Chant est tout aussi simple, d'autant plus que seules les fonctionnalités minimales (création, destruction, synchronisation) sont utilisées. La politique d'ordonnancement du noyau, par exemple, n'est pas exploitée dans Chant.

Au-dessus de cette interface système a été développé le mécanisme d'envoi de messages point-à-point. Concrètement, il s'agit de la gestion des identifiants de processus Chanteurs et des primitives *send* et *receive*. Comme c'est le cas pour les autres environnements intégrant des processus légers avec des outils de communication, seule la réception des messages pose réellement problème. Chant ne fait pas exception à la règle et utilise donc une stratégie de scrutation de l'arrivée des messages par les processus utilisateurs. Cela signifie que chaque processus désirant effectuer une réception de message (*receive*) effectue lui-même la scrutation du réseau, et uniquement pour son propre compte. L'avantage de cette stratégie est d'obtenir une très bonne réactivité du système (les messages sont très vite pris en compte). Son inconvénient est d'effectuer beaucoup de scrutations inutiles dans certains cas.

Le mécanisme d'envoi de messages point-à-point constitue l'outil de base de la plate-forme Chant. Toutes les fonctionnalités des couches supérieures reposent sur celui-ci. Parmi elles, l'*appel de service distant* est la plus élémentaire. Le principe est celui d'un RPC « léger », tel qu'il est décrit dans la section *Environnements basés sur l'appel de procédures à distance*. Nous n'examinerons donc ici que les détails spécifiques à Chant.

Chaque contexte peut, lors de son initialisation, enregistrer des points d'entrée, c'est-à-dire attribuer des noms symboliques à certaines fonctions contenues dans son code. Dès lors, des demandes de services distants peuvent être initiées de l'extérieur vers ce contexte. Pour chaque requête, la fonction demandée est exécutée soit par un processus spécial (*server thread*) chargé de l'exécution des services (lorsque le service ne contient pas d'instruction bloquante) ou par un processus léger spécialement créé pour l'occasion (dans le cas contraire). Notons que c'est à l'enregistrement des points d'entrée qu'il convient d'indiquer au système si le service est bloquant ou non.

Les paramètres d'une fonction de « service » comprennent les arguments de la demande de service distant ainsi que l'identifiant du processus appelant. Le cas échéant, il sera donc possible de renvoyer un résultat à ce dernier. Toutes ces fonctionnalités font que Chant fournit aux applications un mécanisme d'appel de service distant fonctionnant (au choix) de façon synchrone ou asynchrone, et qui permet de spécifier si la création d'un nouveau processus léger doit être effectuée pour l'occasion.

Grâce à ce mécanisme d'appel de service distant, les primitives associées aux processus Chanteurs (création à distance, attente de terminaison, ...) ont été très facilement implantées. En effet, la plupart du temps, pour effectuer une opération sur un processus Chanteur distant, il suffit de demander l'exécution distante de la même fonction, mais appliquée au processus léger sous-jacent. Par exemple, la primitive `pthread_chanter_create` effectue un appel de service distant chargé d'exécuter `pthread_create` dans le contexte cible. Bien entendu, l'implantation de toutes les primitives des processus Chanteurs n'est pas aussi immédiate, comme par exemple la primitive `pthread_chanter_join` qui nécessite de bloquer le processus appelant jusqu'à ce que l'appel de `pthread_join` à distance soit achevé.

Enfin, la dernière couche du support d'exécution Chant implante les fonctionnalités de

groupe (Cordes). De façon à garantir que chaque Corde possède un identifiant unique, un processus léger *serveur de nom* est lancé dans le système pour centraliser les demandes de création de Cordes. Par contre, et pour éviter qu'il ne devienne un goulot d'étranglement pour le système, son rôle s'arrête là. En effet, la gestion de la cardinalité des Cordes est assurée par des processus *serveurs de Cordes*, à raison d'un serveur par Corde. Toutes les opérations d'adhésion, de communication ou de synchronisation sont donc effectuées par ces serveurs, qui centralisent toutes les structures de données relatives à la Corde dont ils ont la charge.

Discussion Chant est un environnement distribué fournissant un modèle basé sur des processus légers communicants. *A priori* destiné aux applications parallèles de toutes natures, il est principalement utilisé en tant que cible de compilateurs pour langages data-parallèles. Plusieurs raisons peuvent expliquer cette situation :

- Avant toute chose, le modèle des processus communicants convient bien aux applications data-parallèles. En effet, celles-ci manipulent de larges structures de données qui sont découpées en tranches sur lesquelles il faut effectuer le même traitement en parallèle. Un tel traitement nécessite souvent que des échanges d'informations aient lieu entre les tranches voisines (pour communiquer les données situées « sur les bords »). Un modèle basé sur des processus communicants est donc relativement bien adapté à ce genre d'application.

- De plus, Chant intègre des fonctionnalités spécialement développées pour le support des applications data-parallèles. Les Cordes en sont l'exemple le plus significatif. Elles ont été introduites pour fournir un contexte d'application aux opérations collectives comme la synchronisation par barrières, mais aussi pour faciliter les communications entre processus travaillant sur des « tranches voisines », pour reprendre l'exemple précédent.

- Enfin, l'interface de Chant, en fournissant en même temps des concepts comme l'envoi de message, l'appel de procédure distant, ou encore les extensions des primitives POSIX pthreads à distance n'offre pas un outil de développement facile à maîtriser pour le programmeur d'applications. En effet, celui-ci se retrouve face à une panoplie hétéroclite de modèles de programmation, ce qui pose de gros problèmes de méthodologie de conception. Cet inconvénient, gênant pour un concepteur d'applications, n'existe pas lorsque l'utilisation de l'interface est faite par un compilateur...

En dépit d'une portabilité exemplaire et d'une efficacité honorable, Chant souffre de quelques défauts gênants pour une utilisation en tant que support pour le développement d'applications parallèles irrégulières :

- Comme indiqué précédemment, le modèle de programmation Chant fournit trop de fonctionnalités de natures différentes, ce qui rend la conception d'une méthodologie de développement très difficile.

- Les aspects concernant la régulation de charge ne sont pas traités par Chant. Aussi, même s'il est toujours possible d'y greffer des politiques de répartition de processus lors de leur création, il est difficile d'imaginer comment bâtir une politique de circulation efficace des informations de charge (la préemption n'est pas garantie) au-dessus du support d'exécution.

Actuellement, les travaux des concepteurs de Chant s'orientent vers la conception d'abstractions de plus haut niveau (que celles fournies par Chant) pour le support du parallélisme de données ainsi que vers la conception d'un noyau de multiprogrammation « ouvert » qui permettrait aux utilisateurs de pouvoir le configurer (par métaprogrammation) de manière à répondre le mieux possible à leurs besoins.

4.2.1.3 TPVM

L'environnement TPVM (Threaded PVM) [65, 66, 67] est une extension de PVM (Parallel Virtual Machine) utilisant les processus légers comme unité de calcul. Il est conçu et développé par des chercheurs ayant également travaillé sur le projet PVM, à savoir Adam Ferrari (Université de Virginie, USA) et Vaidy S. Sunderam (Université Emory d'Atlanta).

L'objectif de TPVM est de fournir un support efficace pour la programmation parallèle distribuée qui soit pratique d'utilisation et portable. Étant donné la popularité de l'environnement PVM, les auteurs ont tenu à préserver une certaine compatibilité (du moins en ce qui concerne la philosophie) avec celui-ci, considérant que le paradigme de l'envoi de message était aujourd'hui majoritairement reconnu comme étant la bonne approche.

C'est donc autour d'une approche orientée « processus communicants » très proche de celle de PVM que les auteurs ont conçu TPVM. Au départ, la motivation de ce travail était de pallier à l'inconvénient majeur de PVM, à savoir sa relative inefficacité. Deux aspects étaient particulièrement visés: la gestion des tâches et le recouvrement des communications par les calculs. La gestion de tâches du système PVM s'appuie sur les processus du système UNIX sous-jacent. C'est pourquoi les temps de création et de changement de contexte sont importants, et le nombre de tâches pouvant coexister sur le même processeur est assez faible. Cette dernière caractéristique a aussi pour conséquence que le recouvrement des communications par le calcul est difficile à obtenir.

L'idée était donc de remplacer, de la façon la plus transparente possible, la prise en charge des tâches PVM par des processus UNIX en une prise en charge par des processus légers, dont on connaît les excellentes qualités en matière de performances et de possibilité de recouvrement des communications par les calculs.

Cependant, il serait limitatif de considérer TPVM comme une implantation efficace de PVM. TPVM est un nouvel environnement système, qui, en plus de sa pseudo-compatibilité avec PVM, apporte plusieurs fonctionnalités autorisant l'utilisation de modèles de programmation tels que la mémoire virtuellement partagée ou encore l'exécution dirigée par les données (DataFlow).

C'est précisément ces différents modèles de programmation que nous allons décrire maintenant.

Modèles de programmation Le titre de cette section est au pluriel car TPVM propose en fait trois modèles de programmation de natures différentes:

Envoi de message Comme dans PVM, les tâches TPVM (processus légers) peuvent communiquer entre elles par envois de messages point-à-point. En fait, chaque primitive PVM (`pvm_*`) trouve sa correspondante dans TPVM (`tpvm_*`) qui conserve la même sémantique.

Mémoire virtuellement partagée TPVM fournit un mécanisme s'apparentant à une première étape vers la gestion d'une mémoire virtuellement partagée. Ce mécanisme consiste

en l'accès à distance à des zones de mémoire (appelées *régions*). Ces régions sont typées et peuvent êtres accédées en lecture et en écriture.

Exécution dirigée par les données Il est possible de retarder l'exécution de traitements jusqu'à ce que certaines *conditions d'activation* soient vérifiées. Ce mécanisme rend possible l'exécution dirigée par les données de certaines applications. Il suffit pour cela que chaque traitement soit retardé jusqu'à ce que les données qu'il nécessite en entrée soient disponibles.

Au départ d'une application TPVM, un certain nombre de tâches PVM « hôtes »[7] doivent être lancées sur les différents nœuds de l'architecture. De manière à pouvoir déclencher la création d'une tâche TPVM, (processus légers) sur un hôte, ce dernier doit *exporter* le *comportement* désiré. L'opération d'exportation est simplement une opération d'association d'un nom symbolique à une adresse de fonction locale, suivie d'une diffusion aux autres hôtes. C'est un mécanisme analogue à celui utilisé par Chant.

Comme dans PVM, la création d'une tâche (légère cette fois-ci) peut se faire soit sur un hôte spécifié, soit sur un hôte choisi par le support d'exécution, en fonction de la charge instantanée du système et du comportement désiré. Une fois la création effectuée, le père de la tâche reçoit son identifiant (Tpvm Task IDentifier) qui peut désormais être utilisé pour les opérations d'envoi ou de réception de message. Ces opérations ayant exactement les mêmes sémantiques que celles de PVM, nous n'en parlerons pas davantage ici.

Même si l'envoi de message constitue un mécanisme suffisamment puissant pour exprimer les différentes phases de synchronisation dans les applications parallèles, il n'en reste pas moins un outil délicat à maîtriser, surtout avec des applications massivement parallèles pouvant nécessiter l'enchaînement complexe de milliers de tâches.

Par ailleurs, les architectures hétérogènes sont difficiles à exploiter efficacement car la durée d'exécution d'un traitement peut varier en fonction de la nature du nœud sur lequel il est exécuté. Dans ce contexte, il peut être intéressant d'adopter un modèle d'exécution un peu moins directif que celui de PVM, pour au contraire laisser le soin au système de répartir l'exécution des traitements en fonction de la charge de la machine.

C'est pour pallier à ces inconvénients que les auteurs de TPVM proposent un *modèle d'exécution* des processus *dirigé par les données*. L'idée part du constat que l'exécution d'un processus est très souvent dépendante de la terminaison d'autres processus, dont il utilise les résultats. Un mécanisme qui démarrerait automatiquement un processus lorsque les données qu'il nécessite sont disponibles permettrait d'éviter la programmation explicite de la synchronisation et des échanges d'information, en déléguant cela au support d'exécution. C'est ce mécanisme que TPVM fournit via le mécanisme d'exécution dirigée par les données (figure 4.7), qui présente certaines similitudes avec les mécanismes mis en œuvre dans l'environnement Cilk [13].

La mise en œuvre de ce mécanisme s'effectue assez simplement. Pour exprimer qu'un type de processus (*i.e.* un comportement) nécessite la disponibilité de certaines données pour pouvoir être lancé, il suffit d'enregistrer (via la primitive `tpvm_export`) le nom du comportement associé à une liste de *types* de données (codés par des entiers) dont celui-ci dépend. Cette action donne lieu à la création d'une *règle de dépendance* au sein du système.

D'un autre côté, lorsqu'un processus désire mettre à disposition du système le résultat d'un calcul, il lui suffit d'y transférer ce résultat (via la primitive `tpvm_invoke`) en lui associant

7. Comme nous le verrons plus loin, l'implantation de TPVM s'appuie également sur la plate-forme PVM.

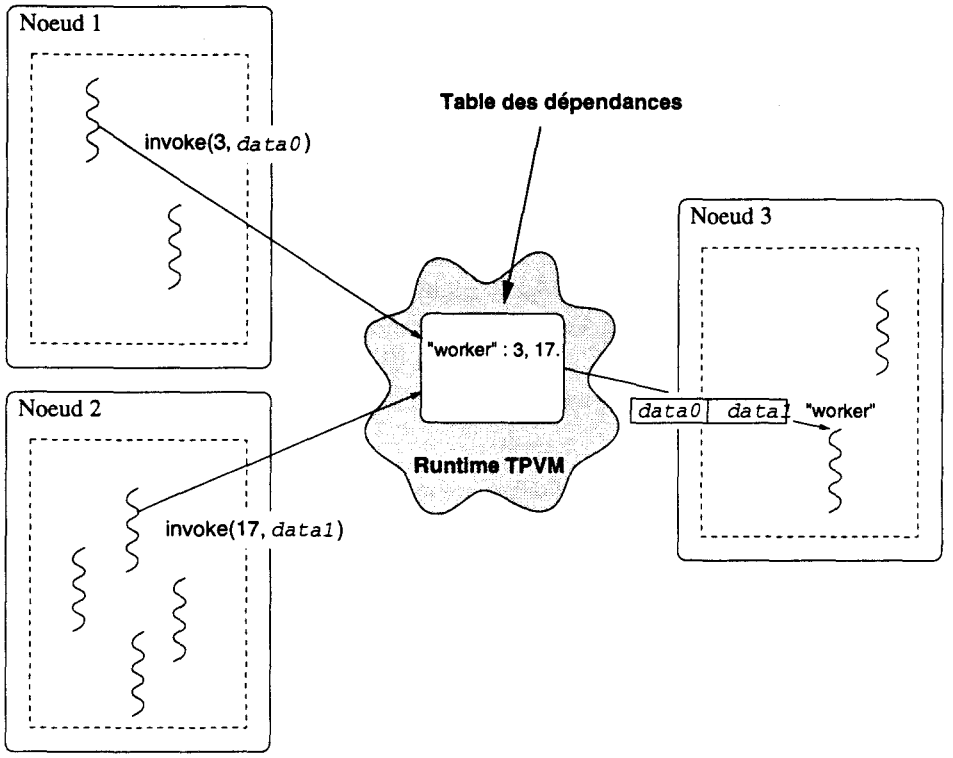

FIG. 4.7 - *Illustration du modèle d'exécution dirigé par les données. Ici, lorsque des données de type* 3 *et* 17 *sont disponibles, un nouveau processus* « worker » *est lancé et peut ainsi les utiliser pour son calcul.*

un type. Cette donnée est alors stockée dans le support d'exécution et associée, de manière non-déterministe, à une règle de dépendance la référençant.

Enfin, lorsqu'une règle de dépendance possède toutes les données qu'elle référence, alors la création d'un nouveau processus a lieu, sur un site choisi par le système (en fonction de la charge instantanée du système), et les données lui sont envoyées sous forme de messages.

Le modèle d'exécution dirigé par les données, que nous venons de voir, propose une philosophie de conception des applications radicalement différente de celle des processus communicants. Bien que les deux mécanismes ne soient pas incompatibles sur un plan technique, il ne semble pas opportun de mélanger les deux styles au sein d'une même application.

En revanche, la troisième fonctionnalité majeure proposée par TPVM, qui s'apparente plus à un outil qu'à un véritable modèle de programmation, est utilisable aussi bien dans un fonctionnement basé sur des processus communicants que sur une exécution dirigée par les données. Cette fonctionnalité consiste à permettre aux processus d'une application d'effectuer des opérations d'*écriture* ou de *lecture de régions mémoire à distance*.

Les auteurs effectuent une analogie entre ce mécanisme, qui permet en quelque sorte à un nœud d'*exporter* une partie de sa mémoire (*i.e.* de la rendre accessible de l'extérieur), et le mécanisme de déclaration des points d'entrée, qui permet d'*exporter* du code. Lorsqu'un processus désire rendre accessible une portion de mémoire (contiguë et non-structurée) du nœud sur lequel il s'exécute, il lui suffit d'effectuer une opération d'*exportation* en associant un identifiant (nom symbolique) aux attributs de cette région. Les attributs comprennent

l'adresse de début de la portion, le type de ses éléments (les régions sont vues comme des tableaux d'éléments simples : entiers, flottants, ...) et enfin le nombre d'éléments qu'elle contient.

Une région mémoire exportée peut ensuite être accédée à distance (en lecture ou en écriture) par n'importe quel processus possédant son identifiant. Cet accès s'effectue en une seule opération (`tpvm_rcpy`) et consiste en une lecture (resp. écriture) de la totalité de la région vers (resp. depuis) une zone de mémoire locale au processus. Aucun outil de synchronisation particulier n'est fourni par TPVM en complément de ce mécanisme. La seule garantie apportée par TPVM est que ces opérations d'accès mémoire sont toujours effectuées de manière atomique. Notons pour terminer qu'une lecture à distance est plus coûteuse qu'une écriture puisqu'elle nécessite deux envois de messages pour s'effectuer.

Implantation et Performances Contrairement à des environnements stabilisés comme Nexus ou Chant, TPVM est encore un prototype expérimental. Il est actuellement implanté sur trois architectures : stations Sun (SunOs, Solaris), stations Silicon Graphix (IRIX) et IBM RS6000 (IRIX). Ces trois implantions sont interopérables.

Sur chacune de ces architectures, TPVM utilise un noyau de processus légers différent, tout simplement pour des raisons de disponibilité. Cela n'est pas gênant dans la mesure où TPVM n'a pas d'exigences particulières quant aux fonctionnalités de multiprogrammation. En fait, seules les primitives de création/destruction, de synchronisation sur verrous et de changement de contexte explicite sont utilisées. La politique d'ordonnancement des processus légers par le noyau ne doit pas avoir d'importance pour les utilisateurs de TPVM, car le système ne fournit aucune garantie quant à elle.

Pour favoriser le portage sur une nouvelle architecture, TPVM s'appuie sur une interface intermédiaire aux fonctionnalités de multiprogrammation. C'est cette seule interface qui doit être réécrite lors d'un nouveau portage.

En ce qui concerne l'accès au réseau de communication, l'implantation repose principalement sur la bibliothèque PVM, qui n'a d'ailleurs pas été modifiée pour des raisons de portabilité. Celle-ci est sollicitée à chaque fois qu'un envoi de message doit être effectué par TPVM (communications entre processus, création à distance, opérations mémoire distantes, ...). Cependant, en raison du risque de « préemptivité » du noyau de communication sous-jacent, l'accès aux primitives PVM se fait en stricte exclusion mutuelle. La conséquence directe est que les primitives bloquantes ne peuvent pas être utilisées et donc que la réception de messages doit se faire par scrutation périodique. Dans TPVM, cette tâche est effectuée sur chaque hôte par un processus léger spécial « démon » chargé de la réception des messages provenant de l'extérieur (figure 4.8).

Les informations globales au système (noms globaux, dépendances de données, ...) sont centralisées par un processus particulier du système : le *TPVM thread server* (figure 4.8). C'est aussi par lui que transitent les requêtes de création de processus dont on laisse au système le choix de l'hôte d'accueil. Notons que, pour l'instant, la seule stratégie de répartition de charge implantée est une stratégie de répartition cyclique sur l'ensemble des nœuds.

Les mesures de performances de TPVM rapportées par les auteurs concernent plusieurs aspects. Le premier aspect concerne la création de processus. Dans ce domaine, les performances de TPVM sont évidemment supérieures à PVM, mais pas de manière écrasante : TPVM est « seulement » trois fois plus rapide pour une création de processus à distance.

Le deuxième aspect des mesures concerne le coût occasionné par TPVM lors des com-

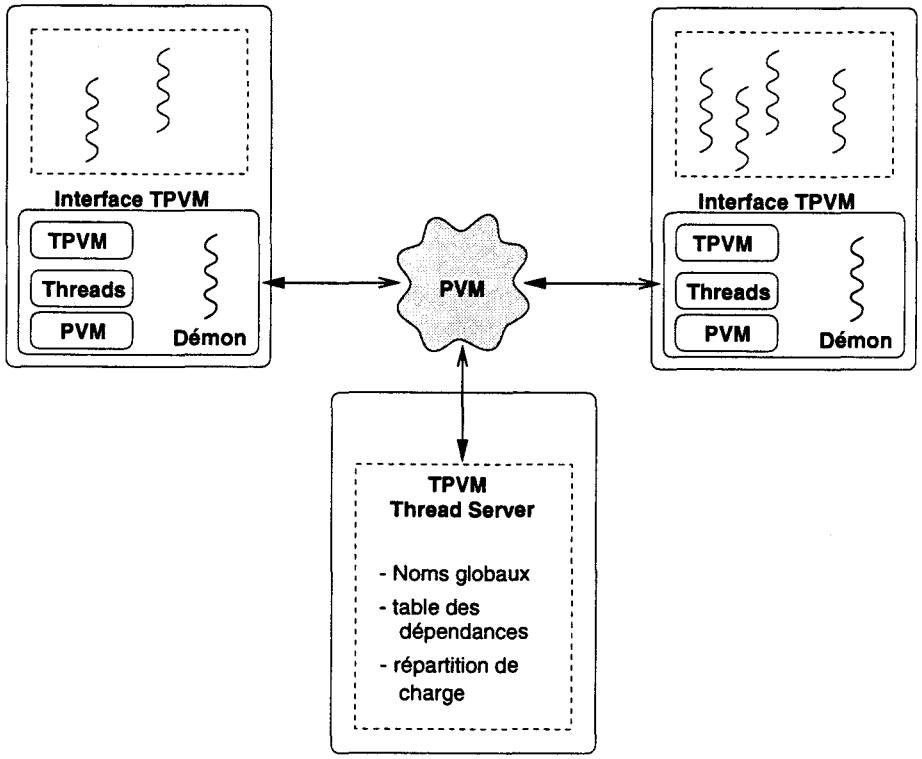

FIG. 4.8 - *Le support d'exécution TPVM.*

munications entre processus. Les communications locales (entre processus légers sur le même hôte) sont évidemment très rapides, mais encore une fois beaucoup moins que l'on pourrait l'espérer (plusieurs millisecondes pour l'envoi d'un message d'un kilo-octet). Cela est dû au mécanisme de recopie systématique des données lors d'un envoi de message, même local. C'est la contrepartie à payer au fait que TPVM fournit une transparence totale de la localisation au travers de son mécanisme d'envoi de message. Les communications à distance, comme on pouvait s'y attendre, ajoutent un surcoût de temps de traitement par rapport au temps de communication PVM. Ce surcoût n'est malheureusement pas constant, sans pour autant être linéaire.

Enfin, un dernier aspect concerne la comparaison des performances de TPVM par rapport à celles de PVM avec quelques applications régulières (multiplication de matrices, etc.). Dans les deux cas, les applications sont développées suivant un modèle d'exécution strictement basé sur l'envoi de messages. Les auteurs parviennent à montrer que le recouvrement des communications par du calcul peut être avantageusement obtenu avec TPVM. Dans ce cas, les performances sont supérieures à celles de PVM et le gain s'est même avéré être de 21 % avec des applications communiquant beaucoup. En revanche, les auteurs précisent avoir obtenu de mauvais résultats (contre-performances) avec des applications régulières très synchrones (modèle SPMD, Single Program Multiple Data), pour lesquelles il n'était pas possible de recouvrir les communications par du calcul et pour lesquelles le système PVM s'avérait parfaitement adapté.

Discussion Le support d'exécution TPVM est un environnement de programmation parallèle « multiparadigme » pour architectures distribuées. Les modèles d'exécution qu'il propose reposent tous sur les processus légers, mais fournissent au programmeur des fonctionnalités de natures différentes. Ainsi, au modèle d'exécution « de base » qui s'apparente à celui des processus communicants (compatible avec PVM), s'ajoutent un modèle d'exécution dirigée par les données (proche du concept Dataflow) et un modèle d'échange d'informations par accès mémoire à distance.

Bien que « composables » entre eux (pas de difficulté technique), ces trois modèles de programmation ne nous semblent pas utilisables simultanément dans des applications de taille conséquente. En l'absence de méthodologie d'utilisation de l'un des modèles plutôt qu'un autre, le programmeur se retrouve avec une « boîte à outils » lui permettant certes de coder (moyennant parfois un effort de programmation non-négligeable) la plupart des applications parallèles courantes, mais ne fournissant pas de « mode d'emploi » permettant de choisir le meilleur outil de structuration en fonction du problème à résoudre. En ce sens, TPVM est comparable à Chant qui ne fournit pas non plus aux utilisateurs un guide méthodologique de conception des applications parallèles.

Parmi les trois paradigmes de programmation fournis par TPVM, l'envoi de message présente un intérêt tout particulier, puisqu'il reprend les mêmes outils (syntaxe et sémantique des primitives) que PVM. Il est d'ailleurs assez facile de « convertir » une application écrite pour PVM en une application utilisant TPVM, par quelques manipulations syntaxiques élémentaires et un effort de regroupement des variables globales en structures de données allouables dynamiquement par un processus léger. En théorie, les nombreuses applications actuellement écrites en PVM pourraient ainsi bénéficier, grâce à un portage vers TPVM, de l'efficacité inhérente à l'utilisation d'une multiprogrammation légère. En pratique, cependant, il s'avère que toutes les applications ne tireraient pas bénéfice d'un tel portage. D'une part, TPVM, comme les autres environnements présentés dans cette section, n'est pas utile en tant que support pour des applications dont le degré de parallélisme est faible (gros grain de calculs). D'autre part, comme nous l'avons vu en examinant les performances de TPVM, les applications parallèles à fonctionnement fortement synchrone ne donnent pas l'occasion de tirer parti d'un quelconque recouvrement des communications par du calcul.

L'implantation de TPVM s'appuie sur la bibliothèque de communication PVM et sur différentes bibliothèques de multiprogrammation. La bibliothèque PVM est utilisée telle quelle, c'est-à-dire qu'aucune modification interne n'y a été apportée. De plus, les exigences du support d'exécution quant aux fonctionnalités du noyau de multiprogrammation sont minimales. Ces deux caractéristiques font de TPVM un environnement portable sur un très grand nombre d'architectures distribuées, si ce n'est l'environnement multiprogrammé actuellement le plus facilement portable[8].

En contrepartie, l'efficacité de TPVM est (comme nous l'avons vu) en retrait par rapport aux autres environnements de la même catégorie. En particulier, le fait de pouvoir fonctionner aussi bien au-dessus d'un noyau de multiprogrammation préemptif que d'un noyau non-préemptif entraîne des surcoûts pour certaines primitives du support d'exécution. Par exemple, la construction d'un message dans PVM utilise une notion de *tampon* global « courant ». Dans le contexte de TPVM, il faut s'assurer que chaque processus léger possède son propre tampon « courant ». Si le noyau de multiprogrammation ne donne pas la possibilité d'attacher des données globales à chaque processus léger, il faut stocker les différents

8. bien que pour l'instant, il ne soit porté que sur trois architectures.

tampons dans une table et parcourir cette table à chaque fois qu'une opération concernant la gestion du tampon courant est effectuée. Cet exemple, choisi parmi beaucoup d'autres, illustre bien la difficulté de concilier portabilité et efficacité. Sur ce plan, TPVM représente un choix de conception ayant jusqu'à présent clairement favorisé la première caractéristique sur la deuxième.

La situation est en passe d'évoluer car une large partie des travaux en cours s'oriente justement vers l'amélioration des performances brutes de l'environnement TPVM, notamment en modifiant l'implantation de la gestion des tampons de messages, en changeant la politique de scrutation du réseau et en distribuant la gestion effectuée par le processus « *TPVM thread server* ».

Mais l'évolution la plus intéressante de TPVM concerne les fonctionnalités de répartition de charge. En effet, les concepteurs envisagent l'intégration dans TPVM d'un mécanisme de répartition de charge dynamique par *migration automatique* de processus légers. Cette fonctionnalité, comparée aux mécanismes de migrations de processus lourds, offrirait un contrôle très fin sur la charge des différents nœuds d'une architecture tout en n'introduisant pas de coûts de régulation prohibitifs. En l'absence de tout contrôle sur les noyaux de multiprogrammation employés, l'implantation d'un tel mécanisme nécessitera cependant l'utilisation d'un compilateur capable de générer des « points de reprise » dans le code exécuté par les processus. De cette façon, il sera possible d'interrompre une tâche TPVM sur l'un de ces points, de transférer par messages l'ensemble des informations formant son « contexte » (tâche que seul un compilateur peut effectuer dans ce cas) sur un autre nœud et de redémarrer la tâche dans l'état où elle était avant son interruption.

4.2.2 Environnements basés sur l'appel de procédure à distance

Le concept d'*appel de procédure à distance* (Remote Procedure Call, ou RPC) a été introduit par Andrew Birell et Bruce Nelson en 1983 [12]. Depuis, il a été très largement étudié dans la littérature [158] et de nombreux travaux de recherche ont proposé des modèles qui en sont directement dérivés. L'objectif était d'améliorer le modèle client-serveur traditionnel en le rendant beaucoup plus transparent. En particulier, il s'agissait d'éliminer la désignation explicite du serveur et la gestion des envois de messages du côté du client. Cet objectif est atteint avec les RPC, comme nous allons le voir en examinant leur fonctionnement.

Plutôt que d'adresser une requête (message) à un processus serveur pour obtenir en retour un résultat (par message également), les concepteurs proposent d'appeler une procédure située sur le site serveur de la même manière que se ferait l'appel d'une procédure locale (figure 4.9). En fait, l'appel d'une procédure à distance est *implicitement* transformé en l'envoi d'un message de requête vers le serveur par l'intermédiaire d'une fonction « *souche* »(1). Pendant ce temps, l'appelant est bloqué. Du côté serveur, l'opération duale est effectuée. La requête entrante est *désempaquetée* (2) (*i.e.* les paramètres sont extraits du message) et la procédure est exécutée. Le résultat renvoyé par la fonction est ensuite empaqueté dans un message qui est retourné à l'appelant (3). Une fois parvenus sur le site appelant, les résultats sont désempaquetés dans la pile du processus appelant par une dernière fonction *souche* (4) et celui-ci est débloqué.

Toute la puissance et l'élégance du mécanisme des RPC résident dans les fonctions *souches* qui peuvent être appelées de façon implicite et transparente. L'écriture de ces fonctions peut être générée automatiquement à partir du profil des procédures (appelées *services* dans ce cas), ou peut rester à la charge du programmeur (lorsque la structure des données à transférer

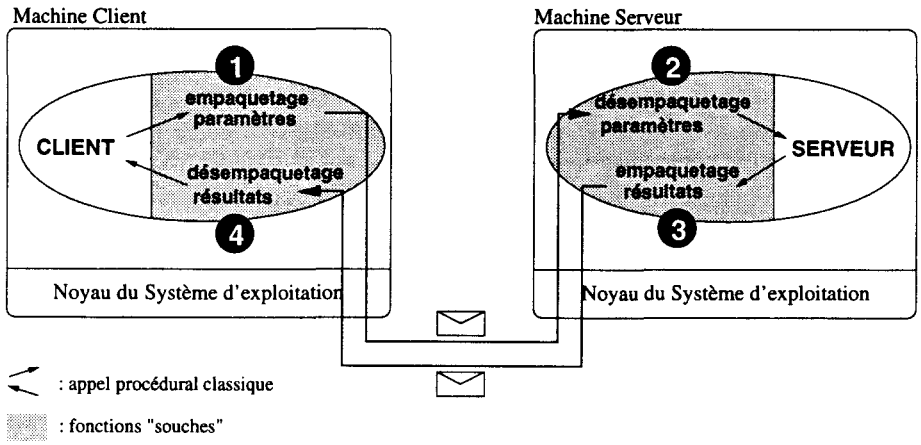

FIG. 4.9 - *L'appel de procédure à distance. Le transfert des paramètres et des résultats est réalisé grâce aux quatre talons (en gris).*

est complexe). Les souches doivent en particulier veiller à assurer d'éventuelles conversions du format des données lorsque la machine client et la machine serveur sont différentes. Il existe d'autres problèmes que le mécanisme doit résoudre, comme par exemple la localisation du processus serveur ou encore le nommage des services, qui sont intéressants à étudier. Cependant ils sortent du cadre de cette section et nous nous contenterons donc uniquement de les avoir évoqués.

L'avantage du modèle est double : la délégation d'un traitement à un serveur peut être réalisée de façon transparente et le code permettant le transfert des données entre les machines n'est écrit qu'une seule fois (alors qu'il est réécrit à chaque appel dans un modèle client-serveur classique).

Si l'on s'intéresse au fonctionnement du processus serveur dans une implantation directe, on s'aperçoit qu'il ne peut répondre qu'à une seule requête à la fois, ce qui est une caractéristique intrinsèque des processus. Dans l'hypothèse où le serveur serait un serveur de fichiers par exemple, c'est-à-dire un processus permettant d'effectuer des opérations de lecture/écriture sur disque, une telle sérialisation des requêtes dégraderait les temps de réponse perçus par les utilisateurs (clients) du serveur.

Pour pallier à cet inconvénient, certains systèmes n'utilisent pas un serveur monolithique, mais au contraire mettent en place un serveur principal « démon » chargé de recevoir les requêtes servant uniquement d'interface aux clients. À chaque fois que ce démon reçoit une requête, il crée un nouveau processus serveur (spécialisé) à qui il délègue l'exécution du service en question. Le démon peut alors de nouveau se mettre à l'écoute d'autres requêtes. Cette approche a le mérite d'exploiter l'éventuel parallélisme sous-jacent de la machine serveur, mais peut s'avérer inefficace pour des services demandant peu de temps de calcul, à cause du coût de création de processus supplémentaires et du temps de recopie des paramètres.

Avec l'avènement des processus légers dans de nombreux systèmes d'exploitation, leur utilisation pour l'implantation de serveurs multiprogrammés s'est vite avérée être une approche très efficace. En effet, ce type d'implantation garde les avantages de celle précédemment évoquée (requêtes traitées en parallèle) et supprime ses inconvénients : les temps de création des processus légers sont moindres et il n'est pas nécessaire de recopier les paramètres puisque les

processus légers partagent tous le même espace d'adressage.

Cette approche, qui introduit les processus légers dans la conception des serveurs de RPC, ne doit pas être confondue avec l'approche qui fait l'objet de cette section, qui consiste à introduire un mécanisme proche de celui des RPC dans des applications multiprogrammées.

La problématique de base est la suivante : « De quelle fonctionnalité majeure faut-il doter un environnement distribué gérant des centaines de processus légers s'exécutant dans des processus différents sur des machines différentes, pour que celui-ci permette une programmation aisée d'applications parallèles efficaces réparties sur plusieurs processus? ».

PM^2 et les environnements décrits dans cette section ont tenté de répondre à cette question en proposant l'utilisation d'une fonctionnalité dérivée de l'appel de procédure à distance, adaptée à un environnement multiprogrammé. Cette fonctionnalité, parfois également appelée RPC par abus, se voit attribuer des noms différents au gré des concepteurs d'environnements qui l'utilisent. Ainsi, ce qui s'appelle RPC dans les environnements Athapascan et DTS, s'appelle LRPC dans PM^2 (Lightweight RPC) ou encore RSR (Remote Service Request) dans Nexus.

Bien que possédant de multiples variantes, que nous examinerons lorsque nous étudierons chaque environnement en détail, le mécanisme d'appel de procédure à distance « *léger* »[9] adopte le fonctionnement général décrit ci-après.

L'appel d'une procédure à distance consiste en la spécification d'un processus lourd cible (pour héberger le processus serveur), d'un nom de service (désignation de la procédure à exécuter) et d'un certain nombre de paramètres (figure 4.10). La prise en charge de cet appel débute sur le site de l'appelant où une fonction souche s'occupe d'empaqueter les arguments dans un message de requête ensuite envoyé au processus cible.

Sur le processus cible, l'arrivée du message de requête déclenche la création d'un processus léger chargé d'exécuter le service demandé. Les paramètres sont ensuite désempaquetés par une seconde fonction souche (selon l'implantation, c'est soit le processus nouvellement créé qui prend cette opération en charge, soit un processus spécial chargé de la scrutation du réseau). Puis le processus léger exécute la procédure. Lorsque l'exécution de la procédure est terminée, le résultat est renvoyé à l'appelant de manière analogue aux RPC traditionnels et le processus léger créé pour l'occasion meurt.

Comme indiqué précédemment, il existe plusieurs variantes à ce mécanisme. En particulier, certains environnements proposent une version dégradée de l'appel de procédure à distance ne comportant pas de phase de retour de résultat. Ce mécanisme, que l'on devrait peut-être plus justement appeler *déclenchement de traitements à distance*, permet d'éviter l'émission du message de retour à la fin de l'exécution d'un service. Le processus appelant n'est donc jamais bloqué lors de l'appel d'une telle primitive. L'utilité de ce mécanisme prend toute son ampleur lorsqu'il s'agit de créer des processus légers « démons » qui, par définition, vont avoir une durée de vie supérieure aux autres processus de l'application.

Dans les sections suivantes, nous allons étudier un certain nombre d'environnements distribués à base de processus légers dont les modèles d'exécution sont tous construits sur le principe de l'appel de procédure à distance. Cette étude n'est pas exhaustive. Seuls les environnements dont le modèle de programmation, les motivations ou encore l'utilisation par les applications présentent un intérêt suffisant seront évoquées.

9. Pour employer le terme des concepteurs de PM^2 ;-)

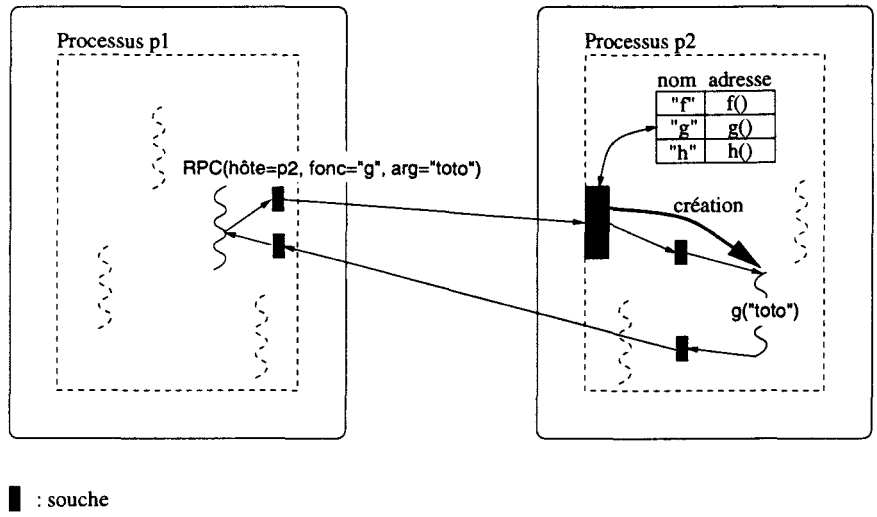

FIG. 4.10 - *L'appel de procédure à distance léger.*

4.2.2.1 Nexus

Nexus [71, 72] est un environnement système issu des travaux de recherche de Ian Foster, Steven Tuecke (Argonne National Laboratory, Argonne) et Carl Kesselman (California Institute of Technology, Pasadena) en 1993. Partis de la constatation que les bonnes propriétés des processus légers leur promettaient un bel avenir dans le domaine du calcul parallèle, Ian Foster et ses collègues ont conçu un support d'exécution multithreadé permettant de supporter l'exécution d'applications parallèles sur architectures distribuées.

En fait, Nexus est plutôt destiné à servir de cible pour des compilateurs de langages parallèles qu'à être utilisé directement par des programmeurs d'applications parallèles [74]. Il est à noter que les concepteurs de Nexus font partie du consortium américain PORTS (« POrtable RunTime System ») regroupant des équipes de recherche universitaires, des laboratoires nationaux et des fournisseurs informatiques. Ce consortium a pour but de définir un support d'exécution commun qui servirait de cible à plusieurs compilateurs pour des langages parallèles (à parallélisme de tâches ou de données).

Nexus se veut être un environnement à la fois portable et efficace. La portabilité de Nexus découle de l'hétérogénéité à plusieurs niveaux supportée par l'environnement. Il est en effet possible de constituer des applications utilisant plusieurs langages de programmation, plusieurs exécutables, plusieurs types de processeurs et plusieurs protocoles réseaux. L'efficacité de Nexus sera discutée dans la section « *Implantation et performances* ».

Bien qu'étant un support d'exécution conçu pour être la cible de compilateurs à parallélisme de tâche, Nexus ne fournit pas l'abstraction d'une machine virtuelle pour les couches supérieures, mais fournit plutôt un ensemble de fonctionnalités pouvant être utilisées en complément des outils standard fournis par le système d'exploitation. Parmi ces fonctionnalités, le multithreading et les outils de communication occupent évidemment une place prépondérante. La manière dont ces deux fonctionnalités ont été intégrées ensemble est au centre de la conception de Nexus [73]. Ce point sera abordé dans la section *Les demandes de services distants*.

L'interface de Nexus est organisée autour de cinq abstractions: les nœuds, les contextes,

les processus légers, les pointeurs globaux et les demandes de services distants (figure 4.11). L'exécution d'une application, que l'on appellera « calcul », consiste en un ensemble de *processus légers* s'exécutant au sein d'espaces d'adressage appelés *contextes*, eux-mêmes alloués sur des *nœuds* de l'architecture sous-jacente. Les processus légers d'un même contexte peuvent communiquer entre eux directement par la mémoire partagée. Les processus légers peuvent également initier des demandes de services distants qui provoqueront l'exécution de procédures dans d'autres contextes en utilisant des pointeurs globaux. Les nœuds, contextes, pointeurs globaux et processus légers peuvent tous être créés et détruits dynamiquement.

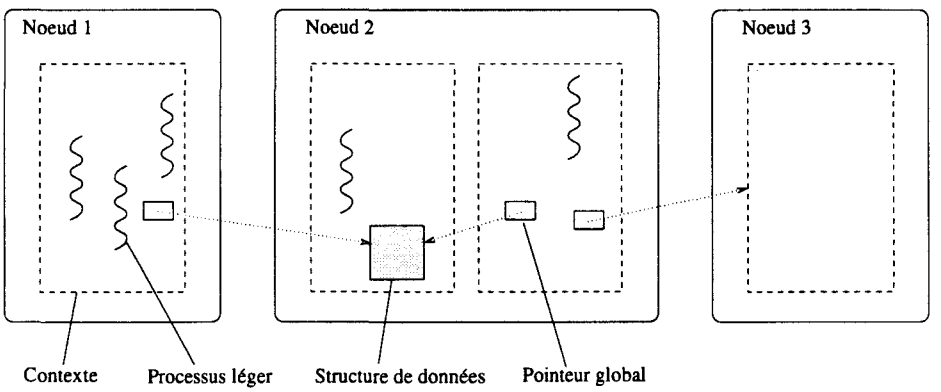

FIG. 4.11 - *Les cinq abstractions dans Nexus. Un processus léger peut initier une demande de service distant dans n'importe quel contexte pourvu qu'il ait un pointeur global le désignant.*

Les processus légers Un calcul est pris en charge par un ou plusieurs processus légers. Les opérations sur les processus légers supportées par Nexus sont les opérations de création, de destruction et de commutation explicite. Les mécanismes permettant une synchronisation à l'aide de moniteurs et de conditions sont également utilisables. Ces mécanismes sont accessibles via une interface constituant un sous-ensemble de l'interface POSIX *Pthread* standard, ce qui permet d'utiliser directement les bibliothèques Pthread disponibles sur de nombreuses architectures.

Les pointeurs globaux Nexus fournit un espace de nommage global de la mémoire via le mécanisme des pointeurs globaux. Un pointeur global est un nom désignant une adresse mémoire donnée dans un contexte donné. C'est une généralisation des pointeurs traditionnels aux architectures distribuées. Un pointeur global peut être communiqué d'un contexte à un autre, ce qui permet par exemple de construire des structures de données réparties sur plusieurs nœuds.

Les demandes de services distants Les pointeurs globaux ont été introduits dans Nexus pour jouer le rôle de « destinations » pour les communications. Ces communications suivent une logique « d'appel de procédure à distance ». Elles sont appelées « demandes de services distants » dans Nexus. Ainsi un processus léger peut, au cours de son exécution, déclencher l'exécution d'un traitement à distance en initiant une demande de service distant. Le contexte cible est désigné par un pointeur global passé en argument au moment de l'appel. La différence

radicale avec les appels de procédures à distance est qu'il n'y a pas de résultat retourné dans le cas d'une demande de service distant.

L'exécution du service distant est prise en charge par un processus léger spécialement créé pour l'occasion. Cela permet de s'affranchir des limitations de code qui contraignent la programmation des services comme lorsque l'on utilise un système à base de messages actifs par exemple. Néanmoins, il est possible de forcer la prise en charge de l'exécution par un processus léger spécialement préalloué par Nexus dans chaque contexte. Cette dernière fonctionnalité, qui exige que le traitement associé au service soit non-bloquant, évite en outre des recopies mémoire dans certains cas, notamment lorsque la couche de communication permet d'accéder directement au contenu des messages.

Les demandes de services distants représentent le seul et unique mécanisme de communication présent dans Nexus, ce qui signifie que les communications directes de processus léger à processus léger, au sens de l'envoi de message traditionnel, ne sont pas supportées. Cependant, il est quand même possible de faire communiquer les processus légers entre eux à l'aide des demandes de services distants, moyennant un effort supplémentaire de programmation (figure 4.12). Rappelons que cela n'est pas un inconvénient dans le cas de Nexus qui a pour principaux « clients » les compilateurs de langages parallèles...

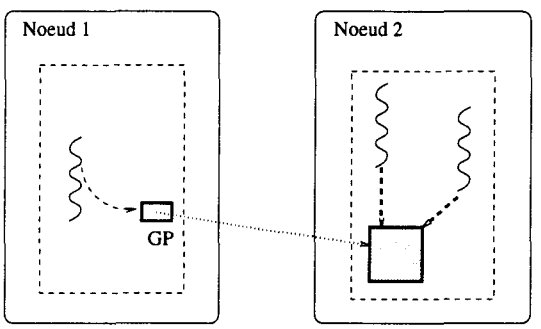

FIG. 4.12 - *La demande de service distant peut permettre aux threads de communiquer. Cette communication s'effectue par le biais d'un pointeur global.*

Implantation et Performances Lorsque Nexus doit être porté sur une nouvelle architecture, il n'est nul besoin de redévelopper des bibliothèques de communication ou de gestion de processus légers puisque Nexus utilise directement les bibliothèques et environnements disponibles sur l'architecture. Ceci est rendu possible par le fait que Nexus n'utilise pratiquement que les fonctionnalités minimales des environnements sur lesquels il est développé. Aussi, ces fonctionnalités, parce que non-spécifiques, sont disponibles sur la plupart des architectures distribuées.

A priori, l'efficacité de Nexus devrait également être bénéficiaire de ce type de démarche puisqu'il est possible d'utiliser les versions propriétaires (optimisées) de certains constructeurs d'architectures pour son implantation. En pratique, cela n'est pas toujours vérifié. En effet, la plupart des bibliothèques propriétaires ne donnent pas la possibilité à l'utilisateur de modifier les sources et de les recompiler, ce qui peut parfois conduire à les utiliser de façon inefficace uniquement pour éviter des conflits avec d'autres bibliothèques. Prenons l'exemple d'une bibliothèque de communication propriétaire et d'une bibliothèque de processus légers

de niveau utilisateur. Si la première n'a pas du tout été conçue dans l'optique d'une utilisation multithreadée, il y a de fortes chances (si l'on peut dire !) pour que celle-ci ne soit pas réentrante. Par conséquent, il ne sera pas possible de laisser un processus léger effectuer un envoi de message pendant qu'un autre effectue une réception de message. En fait, il faut que les processus légers effectuent tous leurs appels à la bibliothèque de communication en exclusion mutuelle. Dans cet exemple, la seule solution consiste alors, pour le processus désirant recevoir un message, à effectuer une scrutation périodique (et donc active) de la file des messages reçus. Comme nous le verrons ci-après, les développeurs de Nexus ont procédé de cette manière sur certaines architectures et ont effectué de nombreux tests de performance pour tenter de dégager les réglages optimaux de la période et de la politique de scrutation en fonction des caractéristiques de l'architecture cible.

Actuellement, Nexus est disponible sur les architectures multiprocesseurs IBM SP2/Aix et Intel Paragon/MP, ainsi que sur les réseaux de stations Sun/Solaris connectées par Ethernet.

Il découle de ce qui précède que l'implantation de Nexus est différente sur chacune de ces architectures. Par exemple sur les Stations Sun, Nexus utilise la gestion des processus légers fournie par le noyau Solaris (Solaris threads), ce qui autorise les processus légers à effectuer des appels bloquants sans danger pour le système. Par contre, sur la machine IBM SP2, la bibliothèque de processus légers utilisée est une bibliothèque de niveau utilisateur. Si un tel processus léger appelle une primitive de communication bloquante, le processus UNIX qui le contient devient bloqué lui aussi. Cette situation est évidemment catastrophique si elle survient alors que d'autres processus légers se trouvent également dans ce processus UNIX. Pour l'éviter, on peut scruter périodiquement le port de communication jusqu'à ce que les données attendues soient disponibles, puis effectuer l'opération de lecture proprement dite qui se déroule alors de façon non-bloquante.

Dans le cas général, les concepteurs de Nexus ont évalué quatre politiques de scrutation différentes :

Réception bloquante Avec un système fournissant un support pour les processus légers au niveau du noyau, il est possible d'affecter dans chaque contexte un processus léger particulier au traitement des requêtes provenant de l'extérieur[10]. C'est l'approche la plus efficace. Elle est malheureusement trop peu souvent disponible.

Scrutation par un processus spécial Dans le cas où il n'est pas possible d'éviter les appels bloquants, on peut affecter un processus léger particulier à la scrutation de l'arrivée de requêtes de l'extérieur. Ici, un surcoût de traitement apparaît, de surcroît dépendant de la politique d'ordonnancement des processus. Aux deux extrêmes du spectre de ces politiques se trouve une politique d'ordonnancement préemptive avec priorités d'une part, et une politique d'ordonnancement non-préemptive d'autre part.

Dans le premier cas, en fixant une grande priorité au processus receveur, on maximise les temps de réponse aux requêtes mais on effectue beaucoup de scrutations inutiles. À l'opposé, une faible priorité introduit un surcoût moins important mais atténue la « réactivité » du système. Un compromis entre les deux est d'ailleurs parfois préférable.

Dans le deuxième cas, le délai de prise en compte d'une nouvelle requête peut fluctuer énormément, puisqu'il dépend uniquement du prochain point de synchronisation que rencontrera le processus qui s'exécute pendant l'arrivée effective de la requête.

10. *i.e.* d'autres contextes

Scrutation dispersée parmi les processus Pour améliorer encore les temps de réponse aux requêtes, on peut forcer les différents processus de l'application à exécuter des instructions de scrutation à certains points de leur exécution. Il suffit pour cela (mais c'est loin d'être trivial) qu'un compilateur insère des instructions de scrutation dans certaines portions de code bien définies. Cette politique rejoint la précédente dans le sens où il s'agit de trouver un compromis entre la réactivité du système et le surcoût introduit par les scrutations inutiles.

Ordonnancement guidé par interruptions La dernière politique de scrutation n'en est pas une puisqu'il s'agit ici de notifier l'arrivée d'une requête en déclenchant une interruption dans le processus cible. Ainsi, il n'y a plus besoin de scruter l'arrivée de requêtes, puisqu'une fonction (un *handler*) sera exécutée sans délai dès qu'une requête arrivera. Séduisant au premier abord, ce mécanisme a deux défauts principaux : d'une part le code de la fonction *handler* ne peut pas comporter n'importe quelle instruction (problème de réentrance) et d'autre part le temps nécessaire à la transmission d'un signal (pour provoquer l'interruption) est souvent plus coûteux qu'un envoi de message de bas niveau.

Ces évaluations montrent que l'éventail des possibilités est large lorsqu'il s'agit d'intégrer des processus légers avec des mécanismes de communication traditionnels. La première idée qui vient à l'esprit après ces constatations est qu'il serait sans doute préférable de disposer de tous ces mécanismes (quand c'est possible) et de laisser les applications (*i.e.* les compilateurs) choisir celui qui lui convient le mieux, de manière dynamique. Les concepteurs de Nexus affirment pourtant obtenir de bons résultats en moyenne avec une politique basée sur la scrutation périodique par un processus léger spécial, et ce quelle que soit l'architecture cible. L'inconvénient majeur rencontré dans ce cas concerne les longs délais de réaction du système lorsque les applications calculent beaucoup et communiquent peu.

Les différentes mesures de performances communiquées à propos de Nexus sont, pour la majeure partie d'entre elles, focalisées sur le surcoût ajouté par le support d'exécution lors des communications distantes. On peut y constater que le surcoût (relatif) introduit par Nexus diminue au fur et à mesure que la taille des données transférées augmente, pour devenir négligeable (inférieur au surcoût introduit par la recopie mémoire des données) pour des données de taille supérieure à 100 000 octets (sur IBM SP2). Cependant, pour des tailles de données de l'ordre de 1 000 octets, le surcoût peut atteindre 50 %. C'est un surcoût important, d'autant plus qu'un bon nombre d'applications parallèles manipulent beaucoup de messages de petite taille (régulateurs de charge, applications d'optimisation combinatoire). Cette caractéristique fait de Nexus un environnement beaucoup plus adapté aux applications de calcul scientifique manipulant de grandes structures de données (calcul vectoriel, imagerie, etc.) qu'aux applications massivement parallèles conçues autour d'un découpage fin des données (optimisation combinatoire, approximation de courbes, etc.).

Applications Nexus a été conçu pour servir de support d'exécution aux compilateurs d'applications parallèles et non pour être utilisé directement par des programmeurs d'applications. On peut donc énumérer une liste impressionnante de compilateurs ou environnements de programmation parallèle :

CC++ (Compositional C++) est un langage de programmation parallèle développé à l'Institut de Technologies de Pasadena [27, 28]. CC++ introduit de nouvelles constructions

pour générer explicitement du parallélisme, ainsi que pour gérer la localisation des objets.

Fortran M est une extension de Fortran à la programmation parallèle modulaire développée au laboratoire national d'Argonne [70]. Fortran M introduit un découpage des applications en modules connectés par des canaux de communication typés. Les communications se font par envoi de messages sur ces canaux. L'architecture cible est virtualisée si bien que le mapping des modules sur les processeurs peut être transparent.

MPICH [86] est une implantation de la spécification MPI (Message Passing Interface) [121] développée au laboratoire national d'Argonne. Cette implantation repose sur une couche logicielle abstraite [84, 85] qu'il suffit d'adapter à diverses bibliothèques de communication de bas niveau pour obtenir autant d'implantations différentes de MPI. Une adaptation à l'environnement Nexus a été réalisée.

nPerl [74] est une bibliothèque de communication pour le langage de scripts Perl.

CAVEcomm est le support d'exécution (pour les communications) de l'environnement de réalité virtuelle CAVE [56].

MTIO (MultiThreading and IO) est une bibliothèque d'Entrées/Sorties multithreadée.

Globus est un projet qui fait suite au projet Nexus (et qui s'appuie dessus) dont le but est de fournir une infrastructure logicielle supportant le calcul parallèle distribué à hautes performances. Globus sera discuté dans la section suivante.

Discussion Comme on peut le constater, l'environnement multithreadé Nexus est utilisé dans de nombreux projets de recherches. De plus, des partenaires industriels (Aerospace Corporation, Trusted Information Systems Inc.) sont impliqués dans son développement.

La Corporation Aerospatiale américaine est partenaire de son développement. Aussi Nexus est-il aujourd'hui un système parfaitement opérationnel et très peu d'environnements à base de threads peuvent revendiquer sa « popularité ».

La notion de pointeur global associée au mécanisme de demande de service distant est un paradigme de programmation qui semble bien adapté à la programmation parallèle. Sa simplicité permet en outre une réalisation concrète *a priori* efficace, puisque toutes les opérations sur ces pointeurs (transmissions, déclenchements de services) sont explicites.

Nexus n'introduit aucune sémantique particulière quant au fonctionnement de l'ordonnanceur des processus légers. Ainsi, d'une architecture à une autre, l'implantation de Nexus pourra varier d'un ordonnanceur non-préemptif à un ordonnanceur préemptif avec priorités. Cette particularité a le mérite de favoriser la portabilité de l'environnement, puisque n'importe quelle bibliothèque de processus légers peut convenir, mais elle peut être gênante pour certaines classes d'applications.

Par exemple, une application contenant un ordonnanceur de charge intégré (sous forme d'un processus léger « observateur du système » sur chaque site) nécessite que chaque site puisse répondre très rapidement aux requêtes des autres sites, pour communiquer sa charge locale. Si l'ordonnanceur de processus n'est pas préemptif, alors cette propriété ne peut être garantie.

De même, une application manipulant une notion de « traitements urgents » ne pourra être implantée de manière satisfaisante si l'ordonnanceur de processus ne gère pas plusieurs

niveaux de priorités pour les processus. Notons que dans le cas où une application se déroule sur une architecture hétérogène, la situation se complique davantage puisque les propriétés de l'ordonnanceur peuvent alors varier d'un nœud à un autre.

Les inconvénients précédents découlent en partie de la volonté de réutiliser au maximum les bibliothèques disponibles sur les différentes architectures visées. Paradoxalement, cette réutilisation maximale ne conduit pas à une portabilité aisée. En effet, nous avons pu voir dans la section « *Implantation et performances* » que différentes politiques de scrutation devaient être évaluées pour la réception des requêtes lors du portage de Nexus sur une nouvelle architecture. Ces évaluations ne sont pas triviales (il faut réécrire des programmes entiers), demandent à être menées avec précision et doivent déceler d'éventuelles incompatibilités entre les bibliothèques utilisées. Ceci fait que le portage de Nexus sur une nouvelle architecture n'est pas forcément une tâche facile.

Citons pour en terminer avec Nexus qu'il s'intègre actuellement au sein d'un projet beaucoup plus vaste nommé Globus. Ce projet vise à définir les bases d'une infrastructure capable de supporter des applications distribuées sur des sites à l'échelle internationale. Ces applications pourront impliquer des centaines de ressources localisées dans de multiples domaines administratifs et pourront utiliser plusieurs réseaux de capacités très différentes. Parmi les différents travaux de recherches menés au sein du projet Globus, certains s'inscrivent dans le thème « *Localisation de ressources et gestion de protocoles* ». Le but général est d'apporter des solutions permettant dans un premier temps de localiser, et dans un deuxième temps d'utiliser au mieux les ressources disponibles dans ces réseaux hétérogènes à grande échelle.

Nexus est la plate-forme d'expérimentation du projet Globus. Il contient d'ailleurs déjà des mécanismes permettant de supporter plusieurs protocoles de communications au sein d'une configuration. Ces mécanismes sont suffisamment flexibles pour que les applications elles-mêmes puissent donner les règles de choix d'un protocole lorsqu'une communication doit avoir lieu. L'implantation utilise des informations attachées à chaque pointeur global (figure 4.13). Ces informations concernent principalement les descripteurs de communications pour les différents protocoles de communications utilisables, ainsi que leurs méthodes attachées. Pour l'instant, l'implantation par défaut sélectionne le premier protocole disponible (trouvé dans un fichier de configuration), mais des recherches sont en cours pour fournir des politiques de sélection « intelligentes » automatiques.

4.2.2.2 Athapascan

Athapascan [34] est un environnement de programmation parallèle dont les objectifs sont très proches de ceux de PM^2. Sa conception s'inscrit dans un projet de recherche intitulé APACHE [140] (Algorithmes PArallèles et régulation de CHargE) mené à l'institut d'Informatique et de Mathématiques Appliquées de Grenoble. Le but du projet APACHE est de développer un certain nombre de techniques, d'algorithmes et d'outils pour supporter des applications parallèles (régulières et irrégulières) de manière efficace. En fait, le projet APACHE recherche le meilleur compromis possible entre la facilité de conception des applications, la portabilité des outils et les performances obtenues. Les architectures visées comprennent les machines multiprocesseurs ainsi que les réseaux de stations de travail. Les thèmes de recherches du projet APACHE vont de la conception d'un noyau exécutif parallèle de bas niveau à l'étude d'algorithmes proposés pour différentes classes d'applications en passant par la conception de répartiteurs de charges et d'outils de visualisation et de déverminage d'applications. Ces

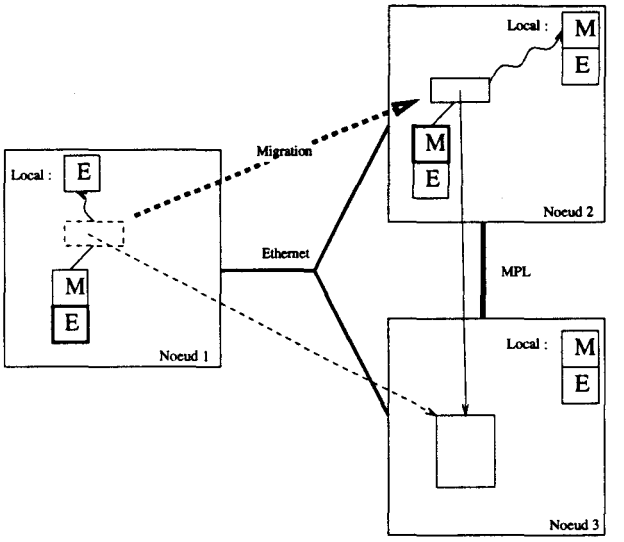

FIG. 4.13 - *Sélection d'une méthode de communication dans Nexus.*

thèmes sont répartis dans les catégories suivantes :

Noyau exécutif Athapascan-0 (Jacques Briat) Athapascan-0 [32, 33] est le support d'exécution du projet APACHE. L'exécution d'une application dans Athapascan-0 se matérialise par un ensemble de processus légers. Les relations entre ces processus légers sont de type « client-serveur ». Les opérations de création ou de communication sont effectuées par un mécanisme proche de celui des RPC (Remote Procedure Calls).

Athapascan-1 et répartition de charge (Brigitte Plateau, Jean-Louis Roch) Athapascan-1 [51, 17] est la seconde couche logicielle du projet APACHE. Elle s'appuie sur Athapascan-0 et offre aux applications une API orientée objet (accessible en C++) ainsi que des mécanismes génériques de répartition de charge. Avec ces mécanismes, une application peut dériver certaines classes d'Athapascan-1 (comme par exemple l'ordonnanceur) ou en instancier d'autres (comme par exemple le graphe de dépendance des tâches) de manière à obtenir l'équilibrage de charge approprié à l'exécution.

Déverminage et Performances (Jacques Chassin de Kergommeaux, Brigitte Plateau) Dans le cadre de ce projet, des outils de déverminage et d'évaluation de performances pour le noyau Athapascan-0 sont développés [64]. L'approche utilisée consiste en la génération logicielle de traces d'exécution. Les travaux de recherche se focalisent en particulier sur les techniques permettant de maîtriser l'intrusion due au traçage, par exemple en corrigeant les perturbations ou encore en déterminisant la réexécution.

Algorithmes et applications (Denis Trystram, Gilles Villard) Ce dernier thème concerne les applications parallèles elles-mêmes. Il consiste en une réflexion sur la façon de concevoir différentes applications réelles avec l'environnement Athapascan. Dans un premier temps focalisées sur le calcul formel et l'algèbre linéaire, les applications traitées se sont récemment étendues à la dynamique moléculaire [9]. Ces travaux, menés en collaboration avec des concepteurs d'applications (mathématiciens, physiciens, automaticiens,

etc.) doivent permettre de dégager des comportements d'applications qui aideront l'amélioration de la plate-forme Athapascan.

Nous allons maintenant présenter plus précisément le support d'exécution Athapascan-0.

Mécanismes de base Le mécanisme de communication dans Athapascan-0 est l'appel de procédure à distance (RPC). Un programme Athapascan se compose de plusieurs *tâches* (*i.e.* processus UNIX) s'exécutant sur une architecture distribuée. Chacune de ces tâches possède un certain nombre de *points d'entrée*, qui sont des fonctions potentiellement[11] appelables depuis d'autres tâches.

L'appel d'une entrée déclenche la création d'un processus léger dans la tâche cible. Ce processus léger reçoit deux paramètres: un tampon d'où il pourra extraire ses paramètres et un tampon où il pourra ranger son résultat. Le processus appelant n'est pas bloqué par un appel d'entrée et il obtient en retour une clé (encore appelée *poignée*) permettant d'identifier l'instance de l'appel d'entrée. Lorsque le processus appelant aura besoin du résultat, il effectuera une opération spéciale d'*attente de résultat* en passant la clé en paramètres. Avec ce mécanisme d'appel/réception en deux temps, il est possible pour un processus de déclencher plusieurs RPC en parallèle tout en continuant sa propre exécution (figure 4.14).

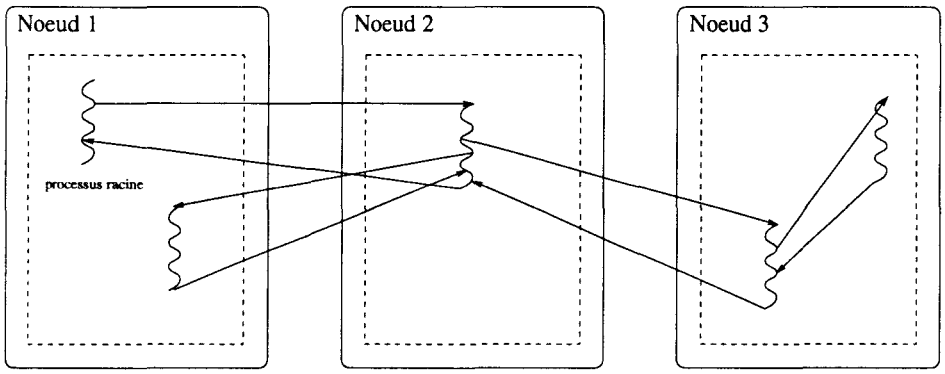

FIG. 4.14 - *Les appels de procédure à distance dans Athapascan-0. La création d'un fils et l'attente de son résultat sont deux opérations disjointes. Notons que les processus d'une application appartiennent toujours à une arborescence unique.*

Modèle de programmation Le modèle de programmation Athapascan est un modèle dit « *à parallélisme de contrôle* ». Ce qui signifie que la décomposition d'un problème en sous-problèmes se fait par un découpage algorithmique, plutôt que par un découpage guidé par les données.

Une application parallèle en Athapascan consiste, au moment de son lancement, en un processus (léger) chargé d'effectuer un certain calcul. Au fur et à mesure, ce processus va découper le calcul en sous-calculs pouvant être effectués en parallèle et va créer autant de processus que nécessaire pour les prendre en charge. Ce processus va se répéter récursivement (les processus nouvellement créés vont eux aussi décomposer leur calcul) jusqu'à ce que des calculs *élémentaires* (qu'on ne peut plus découper) soient obtenus. Lorsqu'un sous-calcul est

11. Un point d'entrée doit être explicitement exporté pour pouvoir être appelé

terminé, il renvoie son résultat au calcul de niveau immédiatement supérieur et le processus qui le prenait en charge se termine. Pour effectuer cette décomposition arborescente, Athapascan fournit deux abstractions permettant d'exprimer la décomposition d'un problème en sous-problèmes : les *procédures parallèles* et les *procédures barrières*.

Une procédure parallèle permet à un processus de découper un calcul en n sous-calculs parallèles de même nature (*i.e.* nécessitant l'appel du même point d'entrée pour être lancés). Il s'agit d'un multi-RPC sur une entrée particulière. Un appel de procédure parallèle nécessite donc une cardinalité (le nombre d'instances de la procédure qu'il faut créer), un ensemble de tâches, un tableau d'arguments (les arguments peuvent être différents pour chaque instance) et bien sûr un numéro d'entrée. Les procédures parallèles introduisent un schéma de mouvement des données $1 \rightarrow N \rightarrow 1$ (figure 4.15).

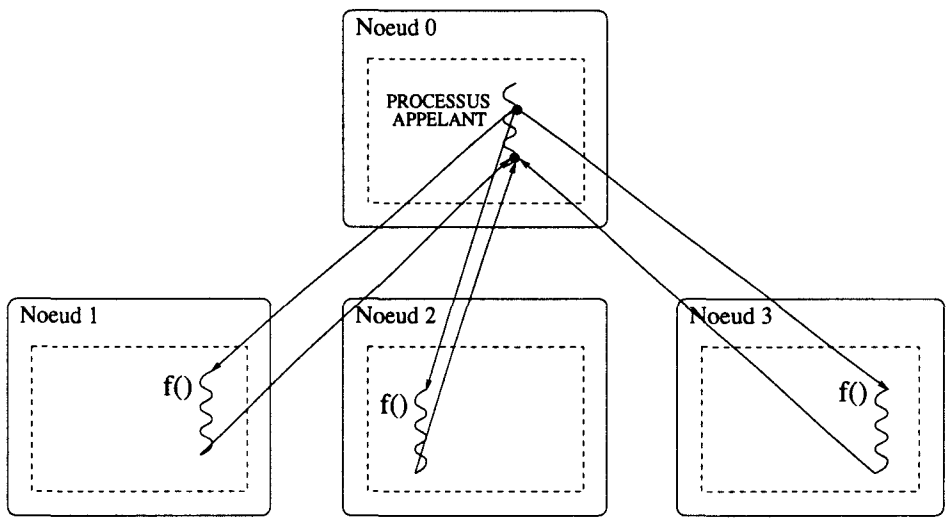

FIG. 4.15 - *L'appel d'une procédure parallèle en Athapascan. Les différentes instances exécutent le même code avec des données différentes.*

Le mode de fonctionnement des procédures parallèles est de type SPMD (Single Program Multiple Data), mais aucune synchronisation entre les différentes procédures n'est effectuée. Cela signifie qu'une instance d'une procédure peut avoir terminé son exécution avant même que les autres instances n'aient été lancées. D'ailleurs, les différentes instances d'une procédure parallèle n'ont pas de moyen particulier pour communiquer entre elles, puisqu'elles n'ont pas l'assurance de coexister.

Pour certaines catégories d'applications, le modèle des procédures parallèles est trop restrictif. En particulier, certains calculs parallèles ont parfois besoin de se synchroniser à certaines étapes de leur exécution soit pour accéder à une ressource partagée, soit pour s'échanger des données. Pour cette raison, les concepteurs d'Athapascan ont ajouté le concept de *procédure barrière*.

Une procédure barrière est un mécanisme destiné à être utilisé par les différentes instances d'une procédure parallèle. Son invocation, qui doit être effectuée par *toutes* les instances, a pour effet de démarrer l'exécution d'un processus sur un point d'entrée particulier. Cette invocation est bloquante tant que toutes les instances ne l'ont pas effectuée. Lorsque c'est chose faite, les instances sont débloquées et reçoivent l'identificateur d'un canal de communication

établi avec le « *processus barrière* » (figure 4.16). De son côté, le processus barrière reçoit autant d'identificateurs de canaux de communication qu'il y a d'instances à synchroniser. Dès lors, des communications classiques, au sens de l'envoi de message traditionnel, peuvent prendre place de façon bidirectionnelle entre chaque instance de la procédure parallèle et le processus barrière.

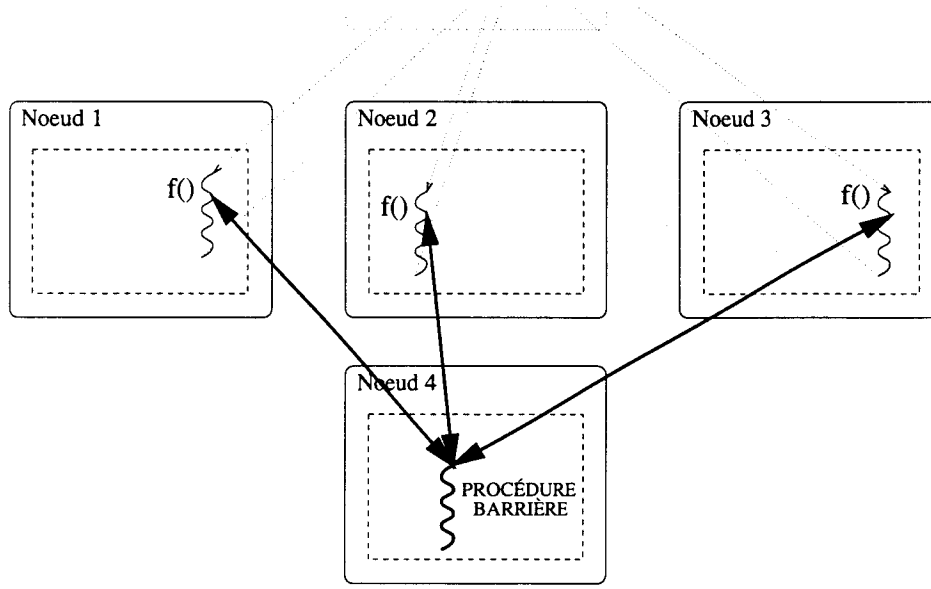

FIG. 4.16 - *L'appel d'une procédure barrière synchronise toutes les instances d'une procédure parallèle. Il est ensuite possible d'échanger des informations via le processus spécialement lancé pour l'occasion.*

Athapascan prône la décomposition de toute application parallèle en procédures parallèles et procédures barrières. Un des objectifs à court terme des concepteurs est de constituer des bibliothèques portables parallèles utilisant ces deux mécanismes de découpage. Pour être portables, ces bibliothèques devront dynamiquement s'adapter à la configuration (nombre de tâches disponibles) et au problème à résoudre. En incluant des règles (*points de choix*) permettant de décider, à chaque fois qu'une décomposition est possible, s'il est « souhaitable » d'effectuer cette décomposition ou s'il vaut mieux résoudre le sous-problème séquentiellement, les concepteurs introduisent la notion de « *poly-algorithme* » adaptatif.

Modèle de calcul La section précédente a introduit la façon dont le découpage parallèle pouvait être effectué par le programmeur. Nous allons maintenant examiner de quelle façon les différents processus d'une application sont pris en charge par le support d'exécution.

Athapascan utilise la multiprogrammation légère pour des raisons d'efficacité et de recouvrement des communications par les calculs. Pour des raisons de portabilité (cf. Nexus) et pour limiter les problèmes de réentrance du code utilisé, l'ordonnanceur de processus Athapascan n'est pas préemptif. Les processus légers se comportent donc comme des coroutines [37], ne cédant le processeur que sur un point de synchronisation. Les avantages et inconvénients de cette politique ont été évoqués dans la section décrivant le système Nexus et seront

repris en détails dans la section décrivant l'ordonnancement des processus dans le système PM^2.

De même, l'ordonnanceur d'Athapascan ne donne pas la possibilité de fixer des priorités aux processus légers créés lors de l'appels de points d'entrée. Il y a deux raisons à cela. La première est que le modèle général d'exécution dans Athapascan ne prévoit pas de gérer différentes priorités d'exécution dans l'état actuel. La deuxième est qu'un mécanisme de priorité est difficile à mettre en place avec un ordonnanceur non-préemptif.

Des études sont toutefois menées au sein de l'équipe pour évaluer l'intérêt de la préemption et des priorités, et la façon dont leur intégration pourrait être faite dans le support d'exécution Athapascan.

Implantation et performances L'environnement Athapascan-0 a été conçu dans un souci de portabilité clairement affiché. Il a donc été développé de manière à pouvoir être porté facilement sur différentes bibliothèques de communications et noyaux de multiprogrammation.

Les différents portages stables d'Athapascan sont actuellement tous basés sur PVM (Parallel Virtual Machine), qui est sans conteste la bibliothèque de communication la plus utilisée au monde et qui présente l'énorme avantage d'être disponible sur de multiples architectures. De plus, certaines architectures disposent d'une version propriétaire de PVM beaucoup plus efficace que la version « générique » de niveau utilisateur. C'est par exemple le cas avec la machine IBM SP2, pour laquelle la bibliothèque *PVMe* (IBM) est disponible et utilisée par Athapascan.

Toujours pour des raisons de portabilité, Athapascan n'utilise que les fonctionnalités minimales des noyaux de multiprogrammation les plus répandus. Cette approche est la même que celle de Nexus. Par conséquent, les avantages et inconvénients sont également les mêmes (voir section précédente sur Nexus). En ce qui concerne l'ordonnanceur de processus, une politique non-préemptive a été choisie, qui conduit à un fonctionnement en « coroutines » des différents processus d'une même tâche. Les arguments avancés par les concepteurs d'Athapascan en faveur de cette politique sont les suivants :

- Une politique d'ordonnancement préemptive introduit nécessairement des changements de contexte « superflus », ce qui est *a priori* [12] coûteux. Ces changements de contexte produisent également des ruptures dans le processeur qui dégradent le fonctionnement des caches mémoire et des pipelines internes.

- Le temps partagé pose des problèmes de réentrance du code, puisqu'un processus peut être interrompu à n'importe quel moment de son exécution. Il faut donc protéger toutes les fonctions non-réentrantes, y compris celles des différentes bibliothèques utilisées.

Comme c'est le cas pour tous les environnements tentant d'intégrer la multiprogrammation avec des bibliothèques de communication existantes, une politique de scrutation du réseau de communication a dû être choisie. En fait, Athapascan en propose deux.

La première consiste à effectuer une scrutation du réseau à chaque point de synchronisation rencontré par les processus (changements de contexte, blocages, terminaisons). Cette politique maximise (autant que possible avec un ordonnancement non-préemptif) la « réactivité » du système en répondant aux requêtes dès que l'occasion se présente, et par conséquent favorise

12. Nous verrons dans le chapitre « Réalisation et performances » (6.3.3.2) des mesures montrant que cela n'est pas toujours vrai.

la création de nombreux processus légers concurrents. Notons que cette caractéristique ne présente pas d'intérêt particulier en l'absence de préemption régulière, sauf celui d'augmenter les chances de recouvrir les communications par du calcul.

La deuxième consiste à effectuer une scrutation uniquement lorsqu'aucun processus léger n'est prêt à s'exécuter. Cette politique, qui est en quelque sorte à l'opposé de la précédente, limite la création du nombre de processus légers concurrents au strict minimum, mais augmente en revanche le « tamponnage » des messages dans la couche de communication.

L'évaluation comparative de ces deux méthodes est en cours d'étude.

Les différentes mesures de performances effectuées sur Athapascan montrent que même avec des applications ne contenant qu'un petit nombre de processus actifs simultanément, il y a bien recouvrement des calculs par les communications, ce qui était un des objectifs principaux de la plate-forme Athapascan-0.

Plusieurs mesures ont également été menées pour déterminer le surcoût exact d'Athapascan en matière de communications par rapport aux bibliothèques natives utilisées. Ces mesures montrent que le surcoût relatif d'Athapascan est important pour des petits messages (jusqu'à 50 % avec PVMe) mais devient moindre (inférieur à 15 %) dès que la taille des données transférées devient grande. Ces mesures reflètent le fait que le surcoût logiciel introduit par Athapascan ne dépend pas de la taille des données transférées. La bande passante maximale d'Athapascan est d'ailleurs égale à celle des bibliothèques natives sur toutes les architectures.

Les mesures indiquent donc que, comme Nexus, Athapascan se révèle être plus adapté aux applications communicant de gros volumes de données plutôt qu'aux applications nécessitant l'échange fréquent de petits messages.

Discussion Athapascan-0 est le support d'exécution sur lequel s'appuient toutes les autres couches du projet APACHE. Le modèle de programmation qu'il fournit est l'aboutissement d'une réflexion guidée non seulement par les besoins des applications, mais aussi par les exigences des outils de déverminage et d'évaluation de performances et surtout par l'interfaçage avec la couche « répartition de charge » Athapascan-1. C'est ce qui explique que l'accent ait été mis sur un style de programmation parallèle simple et bien contrôlé.

Les applications visées par le projet APACHE ont une intersection non-vide avec celles visées par le projet ESPACE. Il est donc intéressant de constater qu'Athapascan et PM^2 convergent vers l'idée commune que le paradigme de l'*appel de procédure à distance* est mieux adapté à ces applications que celui de l'*envoi de message*. Athapascan a étendu ce modèle en proposant un mécanisme de *procédures parallèles* qui est en fait un multi-appel de procédure à distance. Pour faire d'Athapascan un langage simple et sûr, les variantes de RPC sans retour de résultats ont été proscrits.

Certaines applications se décomposant en sous-calculs ayant besoin de coopérer entre eux, il était nécessaire de rajouter des fonctionnalités supplémentaires pour les supporter. C'est à cet effet que le concept de *procédures barrière* a été ajouté. Une procédure barrière permet à un groupe de processus de se synchroniser via un processus spécial avec qui elles peuvent communiquer. Ces communications sont effectuées par envoi de message, ce qui se démarque totalement du modèle de programmation proposé jusqu'alors. Bien que ne posant aucun problème sur le plan de l'implantation du mécanisme, cet « écart » de conduite laisse penser que l'appel de procédure à distance n'est peut-être pas le paradigme le mieux adapté aux applications fonctionnant à l'aide de processus coopérants.

Les mécanismes de procédures parallèles et procédures barrières sont les principaux mé-

canismes de base dans Athapascan et toute application doit être décomposée à l'aide de ces deux opérateurs. L'exécution d'une application en Athapascan peut donc être synthétisée par un graphe orienté sans cycle dont chaque nœud représente soit une opération de division d'un calcul en sous-calculs (appel de procédure parallèle ou barrière) soit une opération de fusion des sous-calculs (synthèse des résultats). Cette propriété est intéressante pour la couche Athapascan-1 qui peut alors manipuler le graphe d'exécution d'une application pour calculer une répartition de charge optimale pour une configuration donnée.

4.2.2.3 DTS

DTS (Distributed Task System) est un environnement de programmation parallèle pour architectures distribuées conçu et développé à l'université de Tübingen (Allemagne) par Tilmann Bubeck, Wolfgang Rosenstiel et Wolfgang Küchlin [19, 20]. L'objectif initial était de fournir un support d'exécution pour la parallélisation d'une bibliothèque de calcul formel algébrique nommée SAC-2 [36]. La bibliothèque parallèle résultante s'appelle PARSAC-2 [106]. Suite à ce travail, les concepteurs ont élargi leur étude à la contribution que pourrait apporter le modèle de programmation DTS au développement d'applications irrégulières sur architectures distribuées.

La caractéristique principale de DTS réside dans son modèle de programmation qui a été recherché aussi simple que possible, de façon à faciliter l'écriture, la mise au point et la correction des applications parallèles sur architectures distribuées.

Modèle d'exécution Une première version de la bibliothèque PARSAC-2 a d'abord été implantée sur des machines à mémoire partagée. Cette version utilise la multiprogrammation légère pour exploiter le parallélisme de l'architecture sous-jacente.

L'approche suivie par les concepteurs a été de tenter de garder le même modèle de programmation pour la version distribuée de leur bibliothèque. Étant donné que les fonctionnalités principalement utilisées étaient la création de processus (*fork*) et l'attente de terminaison de processus (*join*), c'est logiquement un modèle basé sur le paradigme *fork/join* que Bubeck, Rosenstiel et Küchlin proposent à travers l'environnement DTS.

Le modèle de programmation de DTS est basé sur un parallélisme de type SPMD (Single Program Multiple Data). La conception d'applications parallèles avec cet environnement doit être guidée par une stratégie de type « *diviser pour régner* », qui offre, par le biais d'appels récursifs multiples, un moyen simple de paralléliser un algorithme.

Le mécanisme général du *fork/join* est exactement celui d'un RPC pour lequel on souhaite différer (et donc effectuer explicitement) l'attente du résultat (figure 4.17).

En effectuant des appels multiples aux primitives *fork* et *join*, on obtient un fonctionnement similaire aux procédures parallèles d'Athapascan (figure 4.18).

Cependant, contrairement aux environnements tels que Nexus, Athapascan ou PM^2, le placement du processus exécutant le service n'est pas effectué par le programmeur. L'environnement DTS contient un processus spécialisé, le « *DTS load manager* », qui prend en charge le placement des différents processus de l'application. Ce processus « répartiteur de charge » reçoit régulièrement des informations de charge provenant des différents nœuds de l'application, ce qui lui permet, lors d'une requête de type *fork*, de choisir le nœud le moins chargé *a priori* pour accueillir la création du processus.

La présence de ce mécanisme suffit à expliquer que les différents processus légers d'une application ne doivent pas exécuter de code ayant des effets de bord. Les concepteurs de DTS

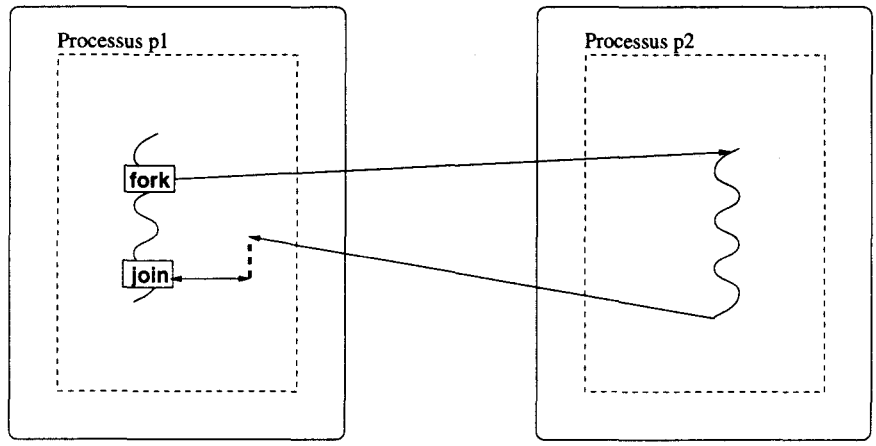

FIG. 4.17 - *Le mécanisme de* fork/join *de DTS. Dans cet exemple, le résultat arrive avant que l'appelant n'ait effectué l'opération* join.

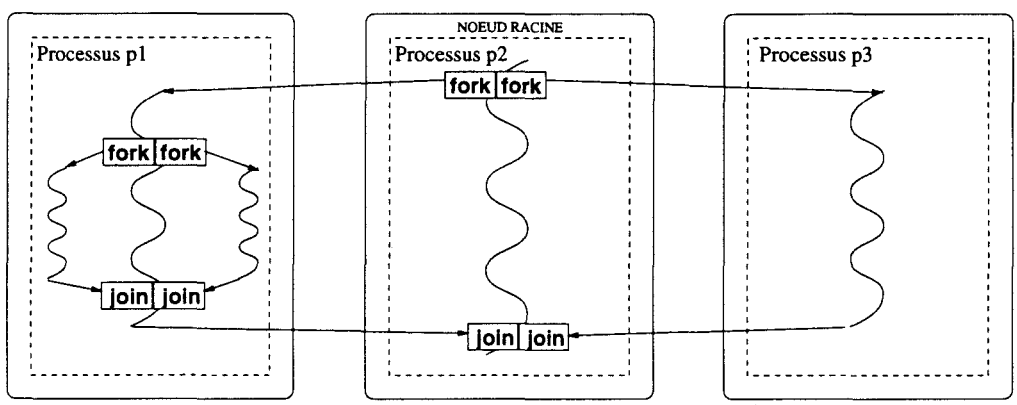

FIG. 4.18 - *Le mécanisme de* fork/join, *utilisé de manière multiple, conduit à une exécution SPMD.*

parlent alors de parallélisme *fonctionnel*.

Toujours dans l'optique d'une amélioration de l'équilibrage de charge entre les différents nœuds, l'environnement peut être paramétré de façon à retarder l'exécution de certaines tâches. L'objectif direct de cette fonctionnalité est de limiter le nombre de processus légers par nœud pour éviter de dépasser un seuil de charge fixé, représenté par une constante de l'environnement. Ainsi, lorsque tous les nœuds d'une configuration DTS contiennent *MAX-JOBS* processus légers actifs, les requêtes de création de processus (*fork*) sont mis en attente dans une file centralisée par le processus « répartiteur de charge ».

Les concepteurs rapportent que cette limitation du degré de parallélisme inter-nœud permet d'atténuer la probabilité d'un fort déséquilibre de la charge lors de phases de terminaison de processus. Nous verrons, lors de l'étude de l'environnement PM^2, que la problématique est différente lorsqu'un mécanisme de régulation de charge par migration est présent.

Implantation et performances L'environnement DTS est actuellement implanté sur les architectures suivantes : réseau de stations Sun (SunOS 4.1.x et Solaris 2.x), multiprocesseurs Solbourne, IBM RS6000, PowerPC (Aix 3.2.x) et Silicon Graphics Indigo (IRIX 5.3). Chaque implantation s'appuie, comme les autres environnements étudiés dans cette section, sur une bibliothèque de communication et sur un noyau de multiprogrammation.

La bibliothèque de communication utilisée est PVM pour toutes les implantations actuelles. Ce choix a été guidé par deux objectifs initiaux : facilité d'utilisation et portabilité très vaste. La contrepartie est que les communications ne sont pas très efficaces, c'est pourquoi les concepteurs indiquent que des travaux en cours concernent une implantation directement au-dessus des *sockets* UNIX.

En ce qui concerne les fonctionnalités supportées par les noyaux de multiprogrammation, l'environnement DTS n'avait pas d'exigence particulière si bien que les bibliothèques « standard » disponibles sur les architectures citées ci-dessus ont parfaitement fait l'affaire. Ainsi, sur les réseaux de stations Sun par exemple, l'interface POSIX-threads fournie par le système Solaris a pu être utilisée.

Bien que le modèle de programmation de DTS permette aux différents processus légers d'une application d'effectuer des opérations distantes comme *fork* ou *join*, les communications PVM ne sont effectuées que par des processus légers spéciaux (les « *node managers* ») implicitement présents dans toute configuration DTS à raison de un par nœud. Cette organisation garantit l'accès aux primitives PVM en exclusion mutuelle (figure 4.19). Les auteurs ne précisent cependant pas la stratégie adoptée par chacun de ces processus pour être en mesure à la fois de scruter le réseau et de se tenir à l'écoute des requêtes provenant des processus locaux.

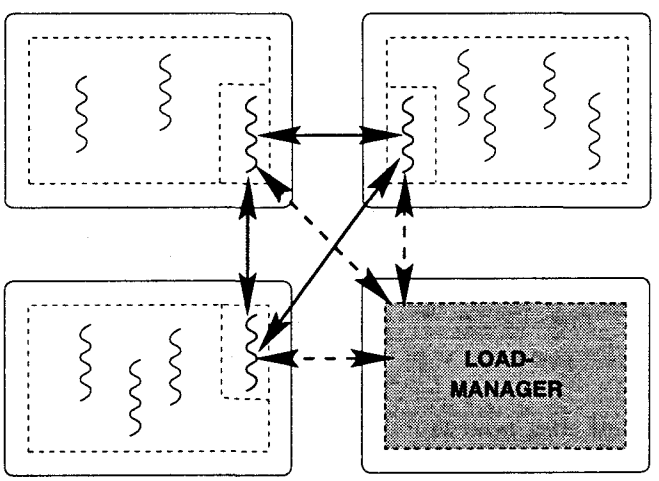

FIG. 4.19 - *Une configuration DTS se compose de plusieurs processus* UNIX *utilisés comme « hôtes » pour les processus légers. Dans chacun de ces processus, un processus léger* « node manager » *sert d'interface réseau. Un processus* UNIX *particulier centralise la répartition de charge.*

La centralisation du mécanisme de répartition de charge dans un processus simplifie grandement son implantation. Cependant, étant donné que chaque requête de création transite par ce processus, les performances générales de l'environnement s'en trouvent grandement al-

térées. Ainsi, le surcoût observé sur des petites applications effectuant des calculs à grain très fin de manière séquentielle atteint 178 % par rapport aux versions programmées directement en PVM. Même s'il convient de relativiser ce surcoût (DTS n'est pas destiné à des applications utilisant un grain de calcul très fin), il faut également s'attendre à une amplification du phénomène dans des applications réellement parallèles, le processus *load manager* risquant alors de devenir un goulot d'étranglement des communications.

De même, il a été maintes fois montré que les algorithmes centralisés n'étaient pas adaptés aux configurations à grande échelle. Cela limite par conséquent l'utilisation de DTS à des architectures constituées de quelques dizaines de machines seulement.

Discussion DTS est un environnement de programmation développé dans le but de porter des programmes parallèles multiprogrammés sur des architectures distribuées. Un effort particulier a été mis sur la simplicité du modèle de programmation. Cet effort a débouché sur le paradigme du *fork/join* distribué. L'intérêt de ce modèle est double.

Avant toute chose, il offre aux concepteurs d'applications parallèles un modèle de contrôle naturel du parallélisme, puisqu'il correspond au modèle bien connu « *diviser pour régner* ». Ainsi est-il relativement simple de paralléliser un algorithme séquentiel récursif bâti sur ce modèle pour l'environnement DTS. De plus, cet opérateur de contrôle étant le seul disponible dans le modèle de programmation, la conception d'applications « algorithmiquement correctes » s'en trouve sensiblement favorisée.

Le deuxième aspect concerne le portage des applications multiprogrammées vers l'environnement DTS. Étant donné que les primitives de base des noyaux de multiprogrammation classique sont les primitives *fork* et *join*, le modèle de programmation proposé par DTS permet de conserver la philosophie générale d'une application multiprogrammée lors de son portage vers DTS. Bien entendu, l'opération de portage n'est pas triviale pour autant. Par exemple, il est souvent tentant (et souvent habile) d'utiliser la possibilité qu'offrent les processus légers à communiquer par la mémoire partagée lors de la conception d'applications multiprogrammées. Le passage sur une architecture distribuée, en l'absence de support pour une mémoire virtuellement partagée, nécessite alors un effort de re-programmation pouvant être assez conséquent.

Outre son modèle de programmation, l'environnement DTS présente plusieurs fonctionnalités intéressantes. L'une d'entre elle concerne la limitation du degré de parallélisme intra-nœud. Ce mécanisme, mis en place pour éviter des déséquilibres de charge trop importants entre les nœuds, a été expérimenté sur de multiples applications par les concepteurs. Les résultats qu'ils avancent sont pour le moins surprenants, puisqu'ils affirment obtenir, sur l'ensemble des tests, les meilleurs résultats avec une limite supérieure (*MAXJOBS*) égale à 5. Les auteurs n'indiquent cependant pas les détails des tests ayant permis d'extraire ce nombre d'or de la plate-forme DTS... Cela étant, nous reviendrons, dans la section consacrée au noyau de multiprogrammation MARCEL, sur l'influence du nombre de processus légers par nœud sur les performances d'une application et nous observerons également des situations étonnantes au premier abord.

Si l'environnement DTS paraît aussi singulier par rapport aux environnements étudiés précédemment, c'est parce qu'il n'est pas à vocation aussi « générale » que les autres. Le fait de ne proposer qu'un modèle de découpage basé sur le *fork/join* enlève à DTS la possibilité de supporter des applications nécessitant l'accès à des fonctionnalités plus élémentaires (RPC asynchrones, etc.). De plus, le fait de proposer une régulation de charge intégrée mais non-

générique à l'environnement ne permet pas aux applications de mettre en place leur propre politique de régulation.

Signalons pour terminer que l'environnement DTS est encore en phase d'évolution. En particulier, des travaux sont en cours pour décentraliser l'algorithme de répartition de charge et pour « virtualiser » la création de processus. Cette dernière fonctionnalité vise à concevoir un mécanisme permettant à DTS de décider, lorsqu'une requête de type *fork* est émise, s'il vaut mieux 1°) créer un nouveau processus sur un nœud distant ; 2°) créer un nouveau processus sur le nœud de l'appelant ; 3°) ou alors faire exécuter le code par l'appelant lui-même. L'objectif est de choisir la solution la plus efficace, compte tenu de la charge instantanée du système. Dans l'état actuel de DTS, ce mécanisme n'est pas disponible, mais il peut être implanté au niveau applicatif, puisqu'il est possible d'appeler la primitive *pthread_fork*, qui crée un processus localement à un nœud, en lieu et place de la primitive normale *dts_fork*.

4.3 Synthèse des environnements existants

La conception d'environnements de programmation parallèle pour architectures distribuées proposant un modèle basé sur les processus légers constitue un domaine de recherche en pleine expansion. Cette importante activité révèle non seulement un intérêt croissant pour le concept des processus légers dans le domaine du calcul parallèle, mais aussi la nécessité de fixer de nouveaux cadres méthodologiques relatifs à l'utilisation de ces derniers en contexte réparti.

Tous les environnements présentés dans ce chapitre ont une vocation affichée pour le support efficace des applications parallèles comportant de nombreux flots d'exécution asynchrones potentiellement parallèles. Tous sont implantés au-dessus du système d'exploitation UNIX, de manière à être utilisables par une large communauté de concepteurs d'applications. Malgré cet objectif commun, ces environnements diffèrent les uns des autres soit par le modèle de programmation qu'ils défendent, soit par l'étendue des fonctionnalités qu'ils fournissent, soit encore par leur adéquation à un domaine d'application particulier.

En dépit de cette diversité, bien peu nombreux sont les environnements présentant un support d'exécution réellement utilisable (tant sur le plan de l'efficacité que sur le plan de l'utilisabilité par un concepteur d'applications) de manière satisfaisante pour le support des applications fortement irrégulières. Nous allons montrer dans quelle mesure au cours de la synthèse suivante.

4.3.1 Modèles de programmation

La constatation immédiate qui découle de l'étude précédente est que deux principaux modèles de programmation émergent des environnements actuels. Ces deux modèles représentent l'extension aux processus légers du modèle des processus communicants (*i.e.* l'envoi de message) et du modèle de l'appel de procédure distante.

En fait, d'autres modèles existent, mais ils restent utilisés de façon anecdotique :

- Le modèle « *Distributed-Pthreads* », mentionné dans l'introduction du chapitre, est partiellement disponible dans l'environnement Chant. Il s'agit de l'extension des primitives classiques associées aux processus légers (création, destruction, synchronisation par structures partagées) de manière à ce qu'elles soient fonctionnelles aussi bien localement qu'à distance. Bien qu'étant le descendant naturel du modèle des processus légers

original, celui-ci reste quasiment inutilisé en raison de son inefficacité catastrophique en milieu distribué. C'est d'ailleurs la raison pour laquelle les concepteurs de Chant n'ont pas réalisé l'extension des primitives de synchronisation dans leur modèle de processus *Chanteurs*.

- Le modèle d'exécution des processus légers dirigé par les données (*Data-Driven Policy*), présent dans l'environnement TPVM de manière optionnelle, est à la base de l'environnement de programmation Cilk [13]. Dans ce dernier, le déroulement d'une application consiste en un graphe de processus non-bloquants dont les arêtes constituent des relations de dépendance. Si ce modèle est particulièrement bien adapté à l'expression du parallélisme dans les langages fonctionnels, il n'en reste pas moins qu'il impose un style de programmation contraignant pour beaucoup d'autres domaines d'application. De plus, son utilisation au travers de langages de programmation classiques (comme le C) nécessite souvent l'intervention de préprocesseurs spécialisés destinés à éviter les lourdeurs d'écriture, ce qui mène parfois à une syntaxe très différente de celle du langage de départ. Enfin, l'implantation d'un support pour un modèle d'exécution dirigé par les données est très délicate en milieu distribué. En particulier, une stratégie consistant à centraliser les informations de dépendance entre tâches révèle vite ses limites lorsque le nombre de tâches devient conséquent (cf. TPVM [66]).

Ceci étant dit, il serait hâtif d'en conclure que seulement deux modèles de programmation sont utilisés en pratique. En fait, les environnements présentés dans ce chapitre utilisent l'un ou l'autre de ces modèles en tant que *modèle de base*, mais chacun en propose différentes variantes ou extensions. C'est ce que nous allons récapituler maintenant.

4.3.1.1 Envoi de message

Les environnements PVC et DTMS adoptent tous deux un modèle de programmation strictement basé sur l'envoi de message point-à-point entre processus légers. Tout échange d'information dans le système s'effectue via ce mécanisme, qui nécessite, de par sa nature, la désignation d'un processus destinataire. Cette caractéristique implique l'existence d'un mécanisme de désignation globale des processus dans tout le système. C'est l'opération de création d'un processus qui se charge d'assigner un nouvel identifiant au processus et de le communiquer au processus ayant initié la création.

Bien que ce modèle de programmation présente de très bonnes propriétés lorsqu'il s'agit de modéliser les interactions entre entités actives du monde réel [130], son utilisation s'avère contraignante dans certains cas, à cause de la lourdeur des mécanismes mis en jeu. En particulier, la désignation systématique d'un processus destinataire lors de chaque transfert d'information n'est pas une contrainte « naturelle » pour les applications parallèles issues de la parallélisation d'un algorithme séquentiel.

C'est pourquoi certains environnements, en l'occurrence TPVM et Chant, proposent des mécanismes complémentaires à l'envoi de message point-à-point. Par exemple, TPVM fournit, outre le modèle d'exécution dirigé par les données évoqué précédemment, un mécanisme d'accès mémoire à distance. Chant, de son côté, propose deux concepts supplémentaires : les appels de procédures à distance et les *Cordes*. Le concept de *Corde* permet de regrouper des processus légers au sein d'entités virtuelles fournissant principalement des services de nommage, de diffusion de messages et de synchronisation.

Cependant, ces fonctionnalités « bonus » fournies par les environnements précédents ne s'apparentent pas à un enrichissement naturel du modèle d'envoi de message, car ils constituent principalement des concepts orthogonaux. De ce fait, le modèle général « mixte » proposé par ces environnements perd les avantages d'une méthodologie de conception simple au profit d'un accroissement du nombre de fonctionnalités disponibles. Notons pour terminer que ces fonctionnalités sont rarement des fonctionnalités « *de base* » de l'environnement et que leur réalisation s'appuie généralement sur le mécanisme d'envoi de message point-à-point.

4.3.1.2 Appel de procédure distante

Les environnements Nexus, Athapascan et DTS proposent tous les trois un modèle de programmation basé sur l'appel « léger » de procédure à distance. Ce mécanisme consiste en la possibilité de déclencher la création distante d'un processus léger pour exécuter un traitement donné (on parle de *service*).

Au premier abord, l'environnement Nexus paraît le plus « rudimentaire » des trois car il est presque exclusivement bâti autour de ce seul mécanisme. En fait, associé à la notion de *pointeur global*, ce dernier constitue une base de niveau suffisamment bas pour autoriser la construction efficace[13] de fonctionnalités plus élaborées. Étant donné que Nexus a principalement été conçu en tant que support d'exécution cible pour des compilateurs, les fonctionnalités « accessoires » n'ont pas été ajoutées. C'est pourquoi il constitue un environnement difficile à utiliser directement pour la conception d'application parallèles.

À l'opposé, la conception des environnements Athapascan et DTS a suivi une démarche consistant à favoriser leur utilisation directe par des programmeurs d'applications. C'est pourquoi ces derniers proposent des mécanismes permettant d'exprimer très simplement le découpage parallèle d'un problème en sous-problèmes.

L'environnement DTS est le résultat d'une approche ayant pour objectif principal de fournir un modèle de programmation des machines parallèles qui soit le plus simple possible. L'idée retenue est d'étendre les primitives *fork* (création de processus) et *join* (attente de terminaison), qui sont fréquemment utilisées par les applications parallèles sur architectures à mémoire commune, dans un environnement distribué. Ce mécanisme, qui constitue le seul et unique moyen d'expression du parallélisme sous DTS, correspond en fait à un appel de procédure distante avec attente différée des résultats.

L'environnement Athapascan fournit, au-dessus d'un mécanisme de base similaire au précédent, la notion de *procédure parallèle* qui s'apparente à un mécanisme de *fork/join* généralisé à plusieurs instances. Les auteurs introduisent la notion d'*équipe* pour désigner les différentes instances contribuant à l'avancée d'un même traitement. L'avantage d'une telle approche est d'augmenter la simplicité de décomposition d'une tâche en sous-tâches tout en conservant un contrôle fort sur cette décomposition.

Contrairement à l'approche « envoi de message », les approches basées sur l'appel de procédure distante permettent une expression plus directe du parallélisme d'une application. En particulier, le mécanisme de RPC permet une extension directe de la méthode « *diviser pour régner* » aux environnements parallèles. Son principal avantage réside dans cette propriété, qui concerne un très grand nombre d'applications parallèles existantes.

En contrepartie, une approche strictement basée sur l'appel de procédure à distance (synchrone) semble ne pas convenir à la construction des applications constituées de plusieurs

13. Il faut tout de même y mettre un bémol car la systématisation du concept des pointeurs globaux peut occasionner une altération des performances due au mécanisme d'indirection sous-jacent.

entités parallèles coopérantes. C'est pour cette raison que l'environnement Athapascan fournit, via le concept de *procédures barrières*, un moyen permettant à plusieurs processus légers (instances de RPC) de se synchroniser et d'échanger des données durant leur exécution.

4.3.2 Régulation de charge

L'obtention d'une exécution *efficace* d'une application parallèle nécessite souvent le recours à des mécanismes de répartition de charge. De plus, lorsque le comportement de l'application n'est pas complètement prévisible avant son exécution, une *régulation dynamique* des traitements permet d'améliorer notablement ses performances.

Aussi, la conception d'un support d'exécution distribué pour applications parallèles est forcément concernée par cet aspect. C'est pourquoi il est intéressant d'étudier les différentes approches suivies à ce propos par les environnements décrits dans ce chapitre. On peut distinguer trois types d'approche :

1. Certains environnements de « très bas niveau », tels que Nexus ou Chant, ne fournissent aucune fonctionnalité spécifique concernant la régulation de charge. Toutes les opérations concernant la création de flots d'exécution sont explicites et nécessitent la désignation d'un processeur cible. C'est donc entièrement au niveau des applications que doit être implanté un éventuel mécanisme de régulation de charge. Étant donné que les applications utilisant directement ces environnements sont des compilateurs, cette démarche ne constitue pas une véritable lacune, mais complique la tâche de ces compilateurs.

2. Plusieurs environnements, tels que PVC, DTMS, DTS ou encore TPVM, fournissent un mécanisme de régulation de charge intégré fonctionnant de manière transparente. Au contraire des environnements précédents, ceux-ci cachent complètement aux applications les problèmes de localisation des traitements sur les différents processeurs de l'architecture. Bien que cette approche aille dans le sens d'une simplification de la conception des applications, elle possède l'inconvénient majeur de représenter une solution figée, qui ne peut en aucun cas prétendre répondre aux besoins de n'importe quelle application parallèle.

3. Enfin, certains environnements sont conçus dès l'origine pour servir de support à des régulateurs de charge implantés de façon séparée. C'est par exemple le cas d'Athapascan-0, qui sert principalement de support d'exécution à la couche *Athapascan-1*. Alors qu'Athapascan-0 fournit des primitives permettant un contrôle explicite de la machine cible, Athapascan-1 comporte un régulateur de charge et transforme de façon transparente la description parallèle d'une application en une série d'appels à la couche Athapascan-0. L'interaction entre les deux couches est bidirectionnelle, car le régulateur d'Athapascan-1 fonctionne en exploitant les différentes informations de charge communiquées par Athapascan-0.

   Une telle organisation présente l'avantage de permettre, le cas échéant, l'utilisation du support d'exécution sans l'intervention d'un mécanisme de régulation de charge préétabli, ce qui donne la possibilité à une application de définir sa propre stratégie de régulation de charge.

Quelle que soit l'approche adoptée, on peut remarquer que tous ces environnements ne fournissent « au mieux » que des mécanismes permettant une régulation par *placement de*

processus, c'est-à-dire que le régulateur ne peut intervenir (pour tenter de rétablir un équilibre global) qu'au moment d'une création de processus dans le système.

Bien que suffisant à garantir l'efficacité d'exécution d'une large gamme d'applications parallèles (*i.e* applications se décomposant en tâches dont on peut estimer les durées d'exécution), l'équilibrage de charge par placement de processus montre ses limites dès que l'on s'intéresse à des applications irrégulières fortement imprévisibles. En effet, l'apparition asynchrone de déséquilibres de charge provoqués par la terminaison ou le blocage de processus est inévitable, ce qui rend un régulateur par *placement* impuissant dans ce cas.

C'est pour cette raison que des environnements comme UPVM[14] [26, 102] ou PM^2 proposent des fonctionnalités permettant la *migration* de processus légers durant l'exécution d'une application. Notons également que les concepteurs de TPVM envisagent l'intégration d'un tel mécanisme dans leur environnement, au moyen d'une approche nécessitant l'intervention d'un compilateur spécialisé. De même, les concepteurs de Chant [44] mènent actuellement des travaux portant sur des mécanismes « de bas niveau » favorisant la conception d'environnements permettant la migration de processus légers. Les auteurs ne précisent cependant pas si ces travaux concernent l'environnement Chant ou s'ils constituent la base d'une nouvelle approche.

4.3.3 Implantations et Performances

Pour la totalité des environnements présentés dans ce chapitre, la portabilité constituait un élément majeur du cahier des charges initial. Cette caractéristique est essentiellement intervenue lors du choix des outils système utilisés pour implanter les mécanismes de communication d'une part, et les mécanismes de gestion des processus légers d'autre part.

Les approches choisies par les différents environnements en matière de gestion des communications sont équivalentes sur le plan de la portabilité. En fait, c'est au niveau des performances et des fonctionnalités qu'elles se distinguent :

Sockets UNIX Certains environnements utilisent directement les *sockets* UNIX pour réaliser les opérations de transfert d'informations entre les différents nœuds de l'architecture. C'est le cas des environnements Nexus, PVC et DTMS. L'avantage de cette approche « bas niveau »[15] est de permettre des communications très efficaces. L'inconvénient est que les fonctionnalités disponibles sont minimales et se réduisent à la possibilité d'émettre ou de recevoir des octets sur un canal de communication. Par conséquent, il peut être nécessaire d'y adjoindre des mécanismes complexes de *tamponnage* des données pour éviter les blocages dus aux saturations éventuelles du réseau de communication.

MPI L'environnement Chant ainsi qu'Athapascan-0b sont basés sur l'interface de communication MPI. Cette approche constitue un très bon compromis puisque l'efficacité est proche de celle des *sockets* tout en offrant des fonctionnalités bien plus élaborées. Cependant, cette approche souffre encore du nombre relativement faible de portages actuels de MPI.

PVM Enfin, les environnements Athapascan-0a, DTS et TPVM sont basés sur la bibliothèque de communication PVM. Cette approche se situe à l'opposé de l'« *approche socket* » puisqu'elle est moins efficace (PVM introduit un mécanisme de tamponnage

14. Nous y reviendrons au chapitre suivant.

15. Encore qu'il soit possible de faire plus bas encore...

occasionnant des recopies de données), mais beaucoup plus souple d'utilisation (configurations dynamiques, support de l'hétérogénéité, etc.). C'est surtout cette dernière caractéristique qui assure le succès de PVM, d'autant plus que la relative inefficacité de la version « standard » est nettement corrigée par des versions « propriétaires » disponibles sur certaines architectures parallèles.

Cette disparité dans le choix des outils de communication n'a pas son équivalent au niveau du choix des bibliothèques de gestion des processus légers. En effet, presque tous les environnements présentés dans ce chapitre n'utilisent que les fonctionnalités minimales relatives aux processus légers (création, destruction, commutation) et se « contentent » donc simplement des bibliothèques disponibles sur les architectures visées.

Hormis l'environnement Chant, qui s'appuie sur l'interface *POSIX Pthreads* dans toutes ses implantations, ces environnements peuvent donc utiliser, sur des architectures différentes, des bibliothèques ayant des interfaces différentes. Si cette approche semble, au premier abord, révéler une certaine indépendance de ces environnements vis-à-vis de la gestion des processus légers, elle n'est pas sans poser plusieurs problèmes:

- Beaucoup de fonctionnalités relatives aux processus légers pouvant s'avérer intéressantes pour le support du calcul parallèle ne sont pas réalisables en dehors du gestionnaire de processus lui-même. Par exemple, un mécanisme de migration de processus léger ne peut pas être implanté sans utiliser certaines fonctions (hibernation d'un processus, relogement mémoire, etc.) propres à la bibliothèque elle-même.

- L'utilisation des fonctionnalités minimales relatives aux processus légers est uniquement guidée par des considérations de portabilité. Cependant, il est important de remarquer que la qualité des fonctionnalités apportées par le gestionnaire des processus a une forte influence sur le modèle de programmation proposé par l'environnement, ainsi que sur ses performances. Par exemple, le fait d'utiliser une bibliothèque ne permettant pas d'ordonnancer les processus de manière préemptive enlève la possibilité de fournir aux applications des garanties élémentaires d'équité quant à l'accès aux ressources de la machine (cf. chapitre 3). De même, il n'est pas possible d'utiliser de mécanismes de gestion d'exceptions (comme en C++) dans les applications si le gestionnaire des processus légers n'est pas « prévu pour ».

- L'utilisation de plusieurs bibliothèques différentes mène souvent à l'utilisation de politiques d'ordonnancement qui sont différentes selon les architectures. Nous avons vu, lors de la description détaillée de l'environnement Nexus, combien il était difficile de mettre au point[16] l'algorithme de scrutation du réseau en fonction de la bibliothèque utilisée.

- L'utilisation de bibliothèques de niveau noyau sur certaines architectures et de bibliothèques utilisateur sur d'autres conduit à des différences de comportement qui amplifient le phénomène précédent.

- L'utilisation de plusieurs bibliothèques différentes multiplie les risques d'utiliser une bibliothèque « boguée ». L'expérience montre en effet qu'en utilisant une même bibliothèque sur plusieurs architectures, on favorise l'apparition d'éventuels bogues (donc leur correction).

16. pour ne pas dire « régler »

Nous examinerons, dans le chapitre consacré à l'implantation de PM^2, notre point de vue sur ces problèmes et la proposition que nous faisons pour contribuer à leur résolution.

En ce qui concerne les performances générales de ces environnements, on peut remarquer qu'elles dépendent essentiellement du modèle de programmation plutôt que des bibliothèques utilisées pour leur implantation. En effet, même s'il est vrai qu'un environnement est généralement plus performant en utilisant directement les *sockets* UNIX plutôt que PVM, les différences constatées sont beaucoup moins flagrantes qu'entre un environnement centralisant toutes les requêtes de création de processus (DTS, TPVM, DTMS) et un environnement utilisant un contrôle complètement distribué (Nexus, Chant, Athapascan). C'est pourquoi il est beaucoup plus important, dans la phase de conception d'un tel environnement, de mettre en œuvre le modèle de programmation le mieux adapté possible aux applications visées avant même d'évaluer les retombées de certaines optimisations.

4.3.4 Conclusion

La difficulté de conception d'un environnement distribué pour le support d'applications parallèles réside dans les délicats compromis qu'il faut adopter entre les *fonctionnalités de base* fournies, la *portabilité* de la plate-forme et l'*efficacité* intrinsèque du support.

Le compromis **fonctionnalités/portabilité** repose principalement sur le degré d'exploitation des possibilités offertes par une architecture. Indirectement, il concerne également l'utilisation des bibliothèques disponibles. Par exemple, la plupart des environnements décrits dans ce chapitre n'utilisent que les fonctionnalités *minimales* relatives aux processus légers, ce qui favorise grandement leur portabilité. À l'inverse, l'utilisation d'une bibliothèque de processus légers fournissant des fonctionnalités évoluées (migration, extension de pile) risque de n'être disponible que sur un très petit nombre d'architectures. Nous verrons, dans le chapitre consacré à l'implantation de PM^2, qu'il faut fortement nuancer cette dernière affirmation.

Le compromis **fonctionnalités/efficacité** est probablement le plus délicat des trois. Il est la conséquence du fait que l'ajout de certaines fonctionnalités a parfois des répercussions sur le fonctionnement général de l'environnement. Par exemple, l'ajout dans TPVM d'une politique d'ordonnancement préemptive entraîne un surcoût notable lors de l'appel des primitives de manipulation des tampons de communication (voir description détaillée de TPVM). Un autre exemple, plus classique, concerne le surcoût occasionné par le tamponnage des messages dans PVM, comparé à une utilisation directe des *sockets* UNIX.

Le compromis **portabilité/efficacité** concerne principalement l'implantation des environnements et touche plus précisément les éventuelles techniques d'optimisation employées de façon à augmenter l'efficacité de certains mécanismes. Par exemple, l'utilisation de mécanismes d'*entrées/sorties* asynchrones rend plus efficace l'exécution d'un ordonnanceur de processus légers au niveau utilisateur, mais diminue sa portabilité.

Les différents environnements décrits dans ce chapitre constituent tous, en quelque sorte, un positionnement vis-à-vis des trois critères que nous venons d'examiner. La figure 4.20 schématise ces propriétés.

Comme on peut le constater sur la figure, il y a deux grandes « tendances » actuelles :

- Des environnements tels que Nexus ou Chant favorisent nettement l'**efficacité** et la **portabilité** de leur approche, en réduisant les fonctionnalités de base proposées. Comme nous l'avons évoqué, cette stratégie, même si elle conduit à d'excellentes performances « brutes » sur des applications régulières, se révèle difficile à utiliser par un concep-

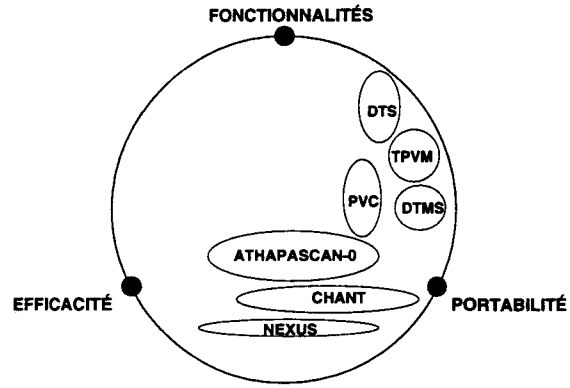

FIG. 4.20 - *La conception d'un support d'exécution représente toujours un compromis entre les fonctionnalités de base fournies, la portabilité de la réalisation et son efficacité intrinsèque.*

teur d'applications (fonctionnalités de bas niveau) mais surtout est insuffisante pour le support efficace d'applications irrégulières (pas de support pour la régulation).

- C'est pour remédier aux problèmes précédents que des environnements tels que DTMS, DTS, PVC ou encore TPVM proposent des mécanismes de régulation de charge intégrés ainsi que des modèles de programmation facilitant la conception des applications parallèles. Ces environnements, au prix d'un sacrifice des performances brutes des mécanismes de base, optent plus volontiers pour un compromis entre **fonctionnalités** et **portabilité**. Nous l'avons vu, cette approche, même si elle s'avère adaptée au support de certaines applications bien particulières, n'est pas satisfaisante dans le contexte général (et exigeant) des applications irrégulières.

Aucune de ces deux tendances n'est donc adaptée au support des applications irrégulières massivement parallèles. Face à ce constat, il est légitime de penser à un meilleur compromis entre les *trois* critères. C'est un peu dans cette lignée que se situe l'environnement Athapascan-0, dont la position vis-à-vis de la régulation de charge est similaire à celle de PM^2. Un des principes essentiels est de fournir des mécanismes de base performants, tout en prévoyant un interfaçage avec une couche logicielle de plus haut niveau qui prendra en charge la stratégie de régulation elle-même.

Néanmoins l'environnement Athapascan-0, plutôt destiné au support des applications à graphe de précédence *semi-prévisible* (cf. section 2.2.3), restreint les fonctionnalités de régulation de charge à des outils d'*instrumentation* et de *placement de processus*. De plus, le choix d'une politique d'ordonnancement des processus non-préemptive limite également certaines implantations d'algorithmes de répartition de charge, comme nous le verrons au chapitre suivant.

On le voit, les environnements actuels de programmation parallèle sont encore mal adaptés au support générique des applications irrégulières à comportement imprévisible. La démarche de conception de l'environnement PM^2, dont l'objectif premier est justement de combler ce manque, s'articule autour de l'apport de fonctionnalités « non-standard » dans le domaine de la programmation à l'aide des processus légers.

Dans les chapitres suivants, nous exposons le modèle de programmation PM^2 en justifiant les différents choix de conception qui en constituent l'originalité. Puis nous montrons comment

les différentes fonctionnalités de l'environnement peuvent être implantées à la fois de façon portable et efficace. Enfin nous illustrons l'intérêt de l'environnement PM^2 en examinant quelques applications l'utilisant ainsi que les performances de l'ensemble.

Chapitre 5

Le modèle de programmation PM^2

5.1 Vue générale de l'environnement PM^2

5.1.1 Rappel des objectifs du projet ESPACE

La conception de l'environnement de programmation PM^2 [53, 131] s'inscrit dans le cadre du projet ESPACE, qui vise à établir un cadre méthodologique et exécutif pour la conception des applications parallèles irrégulières et leur support sur architectures distribuées. L'objectif principal du projet est ambitieux : il s'agit de fournir aux concepteurs d'applications un environnement permettant 1°) d'exprimer simplement le parallélisme intrinsèque à une application et 2°) de supporter efficacement (et de façon transparente) son exécution sur une architecture distribuée quelconque. En fait, l'objectif sous-jacent est d'aboutir à ce que nous appellerons une « *virtualisation totale* » de l'architecture (figure 5.1).

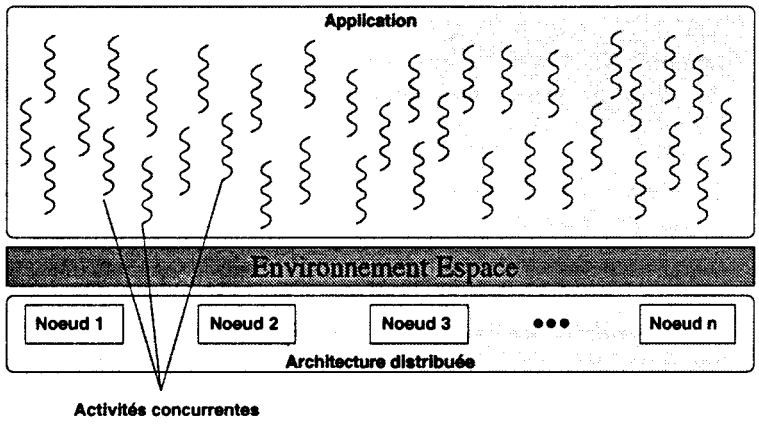

FIG. 5.1 - *L'objectif ambitieux du projet* ESPACE *est de fournir un environnement permettant l'exécution transparente et efficace d'un nombre important d'activités concurrentes à comportement non-prévisible sur une architecture distribuée.*

Nous avons vu au cours des chapitres précédents qu'il serait illusoire de concevoir un tel environnement de façon monolithique, en espérant qu'il supporte efficacement n'importe quel type d'application irrégulière. En particulier, la politique d'affectation des activités aux différents nœuds d'une architecture distribuée doit typiquement s'adapter en fonction du

type de l'application concernée. L'environnement PM², qui constitue le **noyau commun** aux différentes couches logicielles du projet ESPACE, ne doit donc pas imposer une politique d'affectation particulière, mais doit en revanche fournir des abstractions permettant la réalisation d'un tel mécanisme de la manière la plus directe possible.

En somme, il doit constituer **une étape** vers la *virtualisation totale* évoquée précédemment à la fois suffisamment **avancée** pour proposer un modèle de programmation indépendant de l'architecture et suffisamment **souple** pour autoriser un paramétrage fin de la régulation des activités sur les processeurs disponibles.

5.1.2 Les grandes lignes de la « proposition PM² »

Compte tenu des propriétés exhibées par les processus légers pour le support des applications irrégulières (cf. chapitre 3), l'environnement PM² propose un modèle de programmation basé sur l'utilisation de processus légers pour la prise en charge des tâches parallèles d'une application. Dans son principe, une *configuration* PM² consiste donc en un ensemble de processus légers répartis sur les différents nœuds [1] de l'architecture (figure 5.2).

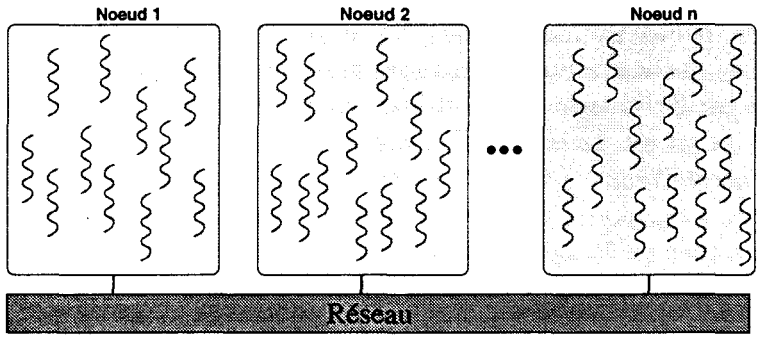

FIG. 5.2 - *Idéalement, le modèle d'exécution proposé par PM² repose sur un ensemble potentiellement très grand de processus légers s'exécutant sur une architecture distribuée.*

De manière à se rapprocher le plus possible de la notion de *virtualisation totale*, le modèle de programmation PM² privilégie trois concepts centraux qui sont la *virtualisation* des processeurs, la *concurrence* des activités et la *mobilité* des activités :

Virtualisation La conception d'une application parallèle comportant un parallélisme massif et irrégulier ne doit pas être guidée par des caractéristiques telles que le nombre de processeurs disponibles sur l'architecture cible. Au contraire, elle doit se focaliser sur l'expression du parallélisme inhérent au problème traité, en *virtualisant* la machine sous-jacente. De même, sa réalisation doit être le reflet le plus fidèle possible de cette spécification, ce qui requiert que le support d'exécution soit capable de prendre en charge un nombre important de flots d'exécution « virtuellement parallèles ». Cette caractéristique, en plus de permettre au système de gérer au mieux les ressources disponibles, est fondamentale en ce qui concerne la *portabilité* des applications sur différentes configurations distribuées.

1. Un *nœud* est un sous-ensemble de l'architecture ne partageant pas de mémoire avec le reste. Il en résulte qu'un nœud peut être monoprocesseur ou multiprocesseurs, mais qu'il ne communique avec les autres nœuds que par un réseau de communication.

Le modèle de programmation PM² propose le découpage parallèle des applications à l'aide du mécanisme d'*appel de procédure à distance léger*, qui assure une expression naturelle du parallélisme tout en rendant transparente la gestion des processus légers sous-jacents.

Concurrence La virtualisation décrite précédemment ne peut être satisfaisante que si toutes les tâches éligibles à un instant donné *progressent simultanément*. Cette propriété, qui est très naturelle dans un système d'exploitation classique (par exemple dans UNIX), l'est tout autant dans une application susceptible de comporter des tâches interactives, ou même des tâches devant s'acquitter de travaux périodiques, tels que les agents d'information de certains régulateurs de charge. En l'absence d'une telle garantie, c'est-à-dire avec un système favorisant arbitrairement l'exécution d'une tâche plutôt qu'une autre, l'*équité* de l'accès aux ressources n'est pas respectée et des situations de famine peuvent se produire (cf. chapitre 3).

Le modèle d'exécution PM² repose sur un partitionnement de l'ensemble des processus légers sur les processeurs de l'architecture, avec un ordonnancement préemptif assurant une exécution réellement concurrente au sein de chacune des parties.

Mobilité des activités Le rôle de PM² n'étant pas de fournir un support d'exécution régulant automatiquement la charge mais au contraire de permettre la construction aisée de régulateurs de charge, les différentes fonctionnalités permettant la création de processus légers sont toutes à « localisation explicite ». Un régulateur de charge construit au-dessus de PM² peut alors, en utilisant des informations fournies par ce dernier, contrôler la distribution des traitements en effectuant un *placement* de tâches *ad-hoc*. Malheureusement, une telle stratégie n'est pas suffisante pour garantir un bon équilibrage de la charge si l'application présente un comportement dynamique irrégulier, car des déséquilibres peuvent apparaître lors de la terminaison de certaines tâches. Seul un mécanisme de *migration* de tâches peut aider à rétablir l'équilibre dans ce cas.

Le modèle d'exécution PM² permet la *mobilité*, c'est-à-dire le déplacement, d'un nœud à un autre, d'une partie des tâches s'exécutant à un instant donné. Pour ce faire, l'environnement PM² fournit un ensemble de fonctionnalités permettant aux applications de commander la *migration* de processus légers pendant leur exécution.

La conception des fonctionnalités proposées au travers du support d'exécution PM² a été effectuée dans une logique guidée par ces trois axes.

5.1.3 Cadre d'utilisation

Pour aboutir à une exploitation des architectures telle que décrite sur la figure 5.2, il faudrait que la réalisation de PM² prenne la forme d'un véritable *micro-noyau* spécialisé dans le calcul parallèle. Cette dernière approche, bien que certainement très bonne sur le plan de l'efficacité, n'entre pas du tout dans le cadre du projet ESPACE qui vise à fournir un environnement utilisable par une large communauté de concepteurs d'applications. De ce fait, l'environnement PM² est conçu comme une couche logicielle au-dessus du système UNIX. De manière à s'approcher le plus possible de la solution « idéale », le modèle d'exécution proposé par PM² repose sur la présence d'un *module* sur chaque nœud de l'architecture distribuée. Ce module, supporté au niveau système par un processus lourd, sert ainsi de « conteneur » pour les processus légers des applications (figure 5.3).

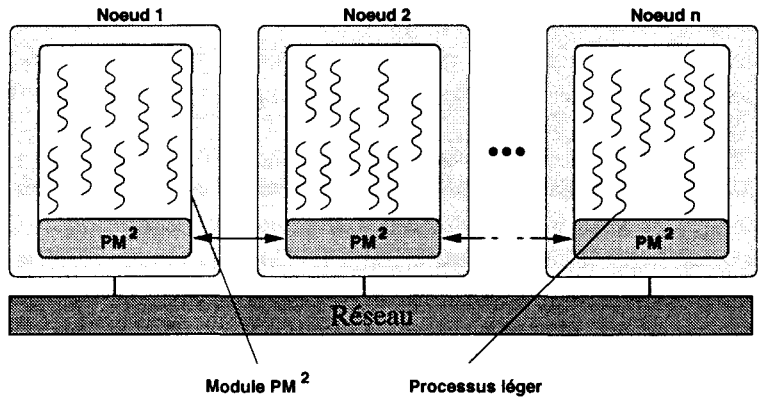

FIG. 5.3 - *Une configuration PM² consiste en un ensemble de modules (typiquement un par nœud) servant de « conteneur » aux processus légers.*

Cette organisation est valable même lorsque certains nœuds de l'architecture sont des machines multiprocesseurs, un parallélisme réel entre les processus légers étant alors exploitable au sein de certains modules. Nous reviendrons sur cet aspect dans le chapitre suivant, consacré à l'implantation de PM², car il est sans incidence particulière sur le modèle de programmation PM².

En particulier, et bien que cela soit possible, nous écarterons l'éventualité du placement de plusieurs modules sur un même nœud. En effet, nous avons vu au chapitre 3 qu'un des intérêts des processus légers était lié à l'efficacité de mécanismes tels que la commutation de contexte entre deux processus. Cet intérêt serait perdu s'il agissait de mettre en compétition deux modules qui, rappelons-le, sont des processus lourds, sur un même nœud.

La « mise en place » effective des modules d'une application sur une architecture distribuée ne revêtant pas d'intérêt particulier du point de vue du modèle de programmation, ce point sera abordé dans le chapitre consacré à l'implantation de l'environnement.

5.1.4 Plan du chapitre

Le cadre d'« utilisation » de PM² étant fixé, nous allons maintenant aborder la description de son modèle de programmation proprement dit.

PM² utilise le paradigme de l'*appel de procédure à distance léger* comme mécanisme central de gestion des activités d'une application. Nous examinerons donc, dans un premier temps, le fonctionnement et les avantages de ce mécanisme dans le cadre spécifique du projet ESPACE.

En introduction, nous avons évoqué l'importance des caractéristiques de l'ordonnancement des processus sur la qualité de la virtualisation procurée par un support d'exécution. Par ailleurs, nous avons vu en conclusion du chapitre 3 l'intérêt *des priorités* dans un système destiné par exemple au support d'applications d'optimisation combinatoire. Nous détaillerons donc, dans un second temps, la stratégie d'ordonnancement adoptée dans PM² et son influence sur le modèle de programmation.

Le mécanisme de création de flots d'exécution qu'est l'appel de procédure à distance et l'ordonnancement de ces flots au sein de chaque module ne constituent pas toujours des outils suffisants pour la programmation d'applications exploitant efficacement l'architecture sous-jacente. En fait, des mécanismes de régulation dynamique de charge sont souvent nécessaires.

L'objectif de PM^2 étant de fournir des fonctionnalités de base pour la construction de tels régulateurs, nous examinerons, dans un troisième temps, sa contribution dans ce domaine.

Dans un quatrième temps, nous introduisons un nouveau mécanisme — le *clonage léger* — issu d'un constat sur quelques « faiblesses » du mécanisme d'appel de procédure à distance et représentant un moyen d'expression du parallélisme mieux adapté à certaines situations. Nous discutons, compte tenu des avantages et inconvénients de chacun, d'une politique d'utilisation conjointe de ce mécanisme avec celui de l'appel de procédure à distance.

Enfin, nous concluons sur le modèle de programmation PM^2 et le modèle d'exécution associé. En particulier, nous soulignons l'originalité de notre approche et évoquons les perspectives de ce travail liées aux exigences d'une virtualisation encore plus grande de l'architecture.

5.2 Appels de procédures à distance légers

Le modèle de programmation PM^2 est basé sur un découpage des applications en un ensemble de tâches et sur la prise en charge de chacune de ces tâches par un processus léger de l'environnement. Ce découpage des applications est réalisé en utilisant le mécanisme d'*appel de procédure à distance léger*, dont nous décrivons la sémantique et la syntaxe dans les sections suivantes.

Il est important de noter que, bien plus que le mécanisme lui-même, c'est son adéquation au support du parallélisme massif et son intégration dans un environnement supportant la mobilité des activités qui constituent l'originalité de notre approche.

5.2.1 Principe

L'appel de procédure à distance est un concept destiné à rendre *transparente* l'utilisation de mécanismes de type *client/serveur* (cf. section 4.2.2). L'objectif est de cacher au *client* les aspects relatifs à la réalisation effective du *serveur*. Pour ce faire, le *client* effectue un appel local « classique » à une procédure spéciale. Cette procédure renferme des mécanismes capables de transformer cet appel local en une requête qui sera émise vers un *serveur* apte à rendre le service en question. Une fois le service rendu, le résultat est émis vers la procédure spéciale qui, après avoir extrait les données reçues, retourne le résultat en utilisant les conventions de passage de paramètres du *client*.

Dans PM^2, le mécanisme utilisé est un mécanisme d'appel de procédure à distance **léger** (encore appelé *LRPC* pour *Lightweight Remote Procedure Call*) et diffère sensiblement du précédent. Il consiste en la possibilité qu'a un processus léger de déclencher l'exécution d'une fonction se trouvant sur un *module* distant, celle-ci étant prise en charge par un nouveau processus léger créé pour l'occasion (figure 5.4).

Lorsqu'un processus léger (p_1 sur la figure) effectue un appel de procédure à distance léger, il fournit principalement un nom de fonction à exécuter (on parle de *service*), des arguments et un emplacement mémoire pour le stockage des résultats. De manière interne, ces paramètres sont alors *empaquetés* dans un message, puis celui-ci est envoyé vers le module cible. Une fois le message reçu par le module, le service à exécuter est identifié et la création d'un nouveau processus léger (p_2 sur la figure) est déclenchée. Ce processus extrait (*désempaquetage*) alors les paramètres du message et commence l'exécution du service. Lorsque l'exécution est terminée, le résultat est *empaqueté* dans un deuxième message, celui-ci est envoyé vers le module source et le processus (p_2) disparaît. Une fois le message reçu par le module source, le résultat en est extrait et est rangé à l'emplacement mémoire spécifié lors de l'appel. Le processus

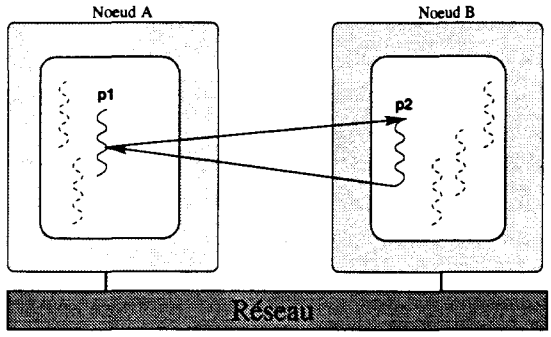

FIG. 5.4 - *L'appel de procédure à distance « léger » (LRPC) consiste en la création d'un processus léger (p_2) dans un module spécifié pour prendre en charge l'exécution d'un service. Lorsque l'exécution du code du service est terminée, le résultat est envoyé au processus appelant (p_1).*

appelant peut alors — nous verrons dans quelle circonstance par la suite — utiliser le résultat de son appel.

Notons que la distinction *client/serveur* n'a de sens que pour une instance de LRPC particulière, car d'une manière générale les différents processus légers d'une application PM² peuvent être à la fois *serveur* d'un LRPC et *clients* de plusieurs LRPC.

L'utilisation d'un mécanisme d'appel de procédure dans un environnement distribué pose deux contraintes :

- Dans un programme classique, la désignation d'une fonction s'effectue par son nom, ce nom étant ensuite traduit en *adresse* (*i.e.* un point d'entrée dans le segment de code) par un compilateur. Dans une application distribuée, constituée de plusieurs programmes distants (les *modules* PM²), cette technique ne peut malheureusement pas être appliquée. En effet, les différents modules d'une application PM² peuvent différer soit par leur contenu (*i.e.* code différent) soit par le format du fichier exécutable (*i.e.* architecture hétérogène). Dans ce cas, l'adresse d'une fonction n'a de sens qu'à l'intérieur de ce module et ne peut en aucun cas être utilisée pour désigner un service distant.

  Il est par conséquent nécessaire de mettre en place un mécanisme de *nommage symbolique* des différents services *exportés* par chacun des modules d'une application. Ce nommage doit être global à toute l'application, de manière à pouvoir identifier les services de manière unique.

- PM² est un environnement destiné à permettre l'exploitation d'architectures hétérogènes telles que des réseaux regroupant par exemple des stations de travail Sun (sous système Solaris) et des PC (sous système Linux). De telles architectures comportent des machines adoptant une représentation différente des données en mémoire. Par exemple, les nombres entiers peuvent être codés sur 2 octets sur certaines machines et 4 octets sur d'autres. De même, l'ordre de stockage de ces entiers peut varier d'une machine à une autre (octet de poids faible d'abord ou octet de poids fort d'abord), etc. Dans un tel contexte, les échanges de données entre les différents modules d'une application ne peuvent s'effectuer de manière brute, mais nécessitent au contraire l'utilisation d'un protocole de conversion de données entre les différents formats.

Il est par conséquent nécessaire d'intégrer un tel mécanisme de « conversion » de formats de données lors des phases de transmission des paramètres et du résultat d'un appel de procédure à distance.

Nous examinerons un peu plus loin la manière dont ces mécanismes sont mis en œuvre dans l'environnement PM^2.

Nous allons maintenant aborder les trois variantes du mécanisme de LRPC disponibles dans l'environnement PM^2, en examinant le domaine d'utilisation de chacune d'elles.

5.2.1.1 Appels synchrones

L'appel de procédure à distance léger **synchrone** est le mécanisme dont la sémantique est la plus proche de l'appel de procédure à distance classique. Lorsqu'un tel appel est effectué, le processus appelant est immédiatement *bloqué* par le système (*i.e.* par PM^2) jusqu'à la disponibilité du résultat.

De par sa nature, ce mécanisme n'augmente pas le degré de parallélisme de l'application. En fait, il peut être vu comme le « déplacement temporaire » d'un flot d'exécution d'un module vers un autre. Son intérêt reste donc limité en tant qu'opérateur de décomposition parallèle. En revanche, la possibilité de « décharger » momentanément un module (et donc un ensemble de processeurs) peut être utile pour rééquilibrer la charge globale sur une architecture parallèle.

En fait, ce mécanisme prend toute son utilité lorsqu'il s'agit d'effectuer des opérations de consultation ou de modification de données locales à un module donné. Dans ce cadre, le but n'est pas de chercher à générer de nouvelles tâches ou à équilibrer la charge de la machine mais simplement d'accéder au contenu d'une *zone mémoire distante*.

5.2.1.2 Appels à attente différée

L'appel de procédure à distance léger **à attente différée** constitue un opérateur de décomposition parallèle majeur dans l'environnement PM^2. Son fonctionnement est comparable à celui d'un appel *synchrone* pour lequel les opérations d'envoi des paramètres et de réception du résultat seraient séparées. Son utilisation s'effectue donc en deux temps :

Appel Dans un premier temps, le processus appelant effectue l'appel en spécifiant les mêmes paramètres que dans le cas d'un appel synchrone (nom de service, arguments et adresse de stockage du résultat) et en y ajoutant une référence sur une variable d'un type particulier[2], qui lui servira de *clé* par la suite. Cette étape donne lieu à la création d'un nouveau processus léger (pour exécuter le service), mais ne « bloque » pas le processus appelant qui peut donc librement continuer son exécution.

Attente Dans un second temps, lorsque le processus appelant désire accéder au résultat de l'appel, celui-ci doit effectuer une opération d'attente en spécifiant la clé qu'il a obtenue lors de l'appel correspondant. Le déroulement de cette opération peut s'effectuer de deux façons, selon le cas. 1°) Le résultat est immédiatement disponible car l'exécution du service s'est terminée entre-temps. Dans ce cas l'opération d'attente est terminée. 2°) L'exécution du service n'est pas terminée, par conséquent le résultat n'est pas encore disponible. L'opération d'attente conduit alors au blocage du processus appelant jusqu'à disponibilité du résultat.

2. Nous illustrerons tout cela sur un exemple dans la section « mise en œuvre ».

Cette décomposition du mécanisme d'appel de procédure à distance est intéressante parce qu'elle permet au processus appelant de continuer son exécution en parallèle avec le processus « *fils* » ainsi créé et donc d'augmenter le degré de parallélisme d'une application. Le système DTS [20], par le biais du mécanisme *fork/join*, fournit une fonctionnalité similaire.

De plus, l'attachement d'une *clé* à chaque instance d'appel à distance permet à un processus de déclencher plusieurs appels en parallèle, puis, le moment venu, d'effectuer les opérations d'attente associées à l'aide des clés correspondantes. Ce mode de fonctionnement est analogue aux procédures parallèles d'Athapascan [33].

Ce mécanisme est particulièrement adapté à la *parallélisation d'algorithmes séquentiels*, car il est alors utilisable à chaque fois que l'on peut déceler un ou plusieurs blocs d'instructions exécutables en parallèle (*i.e.* dont l'ordre d'exécution est sans importance). Dans ce cas, il « suffit »[3] de transformer ces blocs d'instructions en *services* et d'effectuer des appels à attente différée au lieu d'exécuter séquentiellement le code.

Appliquée de manière récursive, cette approche (appelée *diviser pour régner*) devient un puissant outil de parallélisation d'applications. Nous en verrons un exemple typique dans le chapitre 7, en examinant une application d'optimisation combinatoire développée au-dessus de PM².

5.2.1.3 Appels asynchrones

La dernière forme d'appel de procédure à distance léger proposée par l'environnement PM² est dite « **asynchrone** », car elle offre le moyen de créer un nouveau flot d'exécution indépendant du processus appelant. Ce mécanisme, qui s'apparente plutôt à un « déclenchement de traitement à distance », diffère des deux précédents par le fait qu'il ne comporte pas de phase de retour de résultat. En ce sens, il est très similaire au mécanisme de RSR (*Remote Service Request*) proposé par l'environnement Nexus [72].

Son utilisation est destinée à quelques cas de figure bien précis :

- Certaines applications nécessitent l'exécution de traitements en « tâches de fond », ne prenant pas part au calcul principal de façon directe, mais s'apparentant plutôt à des processus *démons* chargés de certaines tâches « système » telles que l'observation de charge de la machine ou encore le ramasse-miettes de la mémoire. La création de ces processus légers particuliers ainsi que certains échanges d'informations entre ces processus peuvent être implantés à l'aide d'appels de procédures asynchrones.

- Nous avons évoqué précédemment l'adéquation des LRPC à attente différée à l'implantation d'algorithmes parallèles de type *diviser pour régner*. Parmi ces algorithmes, beaucoup proviennent de la parallélisation d'un algorithme récursif, chaque appel récursif ayant été remplacé par un appel de procédure à distance. L'exécution d'un tel algorithme donne naissance à une véritable arborescence de processus légers, chaque nœud correspondant à une instance d'appel de la procédure. Dans ce cas, comme dans la version séquentielle, l'algorithme parallèle se termine par une phase de remontée des résultats de proche en proche (plus précisément de fils en père) jusqu'au processus racine.

  Cependant, dans le cas où la procédure initiale est une procédure dite « à récursivité terminale », le déroulement de l'algorithme parallèle devient très maladroit car la phase

3. Nous dresserons le pour et le contre de cette approche en section 5.5

de remontée des résultats est inutile (ils ne sont plus modifiés lors de cette phase). En fait, il est tout simplement inutile de laisser les processus « pères » survivre aux processus « fils », puisque seuls ces derniers auront le dernier mot. Cette optimisation est comparable à celle qui serait effectuée sur la version séquentielle par un compilateur gérant correctement les appels récursifs terminaux. Dans une telle situation[4], les LRPC asynchrones fournissent l'intéressante possibilité de réduire considérablement l'occupation des ressources de la machine.

- Certaines opérations de mise à jour de zones de mémoire à distance peuvent avantageusement être prises en charge par des appels de procédures à distance légers asynchrones. Il s'agit principalement des opérations d'écriture sur des variables ne nécessitant pas une cohérence forte (mises à jour asynchrones). Dans ce cas, l'utilisation de ce mécanisme, comparé à un LRPC synchrone, permet d'éviter un retour de message superflu.

Au premier abord, ce mécanisme paraît suffisamment puissant pour pouvoir supporter l'implantation de diverses stratégies d'échange d'information entre les modules. D'ailleurs, utilisé conjointement avec des mécanismes de synchronisation tels que les sémaphores ou les moniteurs de Hoare, il est possible d'implanter les deux autres formes de LRPC à partir de celui-ci. Après tout, d'un point de vue technique, un LRPC synchrone n'est pas très différent d'un « aller-et-retour » réalisé à l'aide de deux LRPC asynchrones.

Cette idée doit cependant être abandonnée, cela pour au moins deux raisons. La première est qu'une utilisation de l'appel asynchrone destinée à effectuer des échanges d'information entre processus légers revient à un mode de fonctionnement s'apparentant soit à l'envoi de messages (dont on a évoqué les inconvénients en contexte massivement parallèle), soit à une synchronisation des activités par sémaphores distribués (ce qui est encore pire), soit enfin à l'échange d'informations par un mécanisme analogue aux *pointeurs globaux* de Nexus (dont nous avons également évoqué la difficile gestion). La deuxième raison est liée à la mobilité des activités permise par l'environnement PM^2, en d'autres termes au mécanisme de migration des processus. Nous verrons dans une section à venir que PM^2 permet à une application de déplacer un processus léger d'un module à un autre pendant son exécution. Cette possibilité, ô combien intéressante pour la régulation de charge, n'est pas sans causer quelques petits soucis aux mécanismes de LRPC, en particulier en ce qui concerne les retours de résultats. Nous verrons que PM^2 propose plusieurs solutions aux problèmes que l'on devine, mais qu'il ne peut le faire que parce qu'il maîtrise le déroulement des différents types d'appels de procédure légers. Si, par contre, le programmeur utilise des outils de synchronisation ou d'échange d'information de « bas niveau » (nécessaires si l'on ne dispose que de l'appel asynchrone), alors il risque de concevoir une application où il ne sera pas possible de migrer les processus (à cause de multiples contraintes de localité).

Pour ces raisons, l'utilisation de l'appel de procédure asynchrone dans une application ne doit pas être systématique, mais doit, au contraire, être réservée à la réalisation de tâches bien précises, telles que celles évoquées précédemment.

5.2.2 Appels de procédures à distance légers « rapides »

Les trois variantes de l'appel de procédure à distance léger que nous venons de voir ont toutes en commun le fait de déclencher la création d'un nouveau processus léger dans l'environnement. Bien que cette dernière opération ne soit pas intrinsèquement très coûteuse en

4. Que l'on rencontre assez couramment (cf. algorithmes de type Branch & Bound).

terme de durée d'exécution (typiquement quelques dizaines de microsecondes) comparativement aux opérations de communication associées (voir chapitre 6.4.3.4), elle a tout de même un coût sensible en ce qui concerne l'utilisation de la mémoire du module cible.

Pour la prise en charge de « petits » traitements (en terme de ressources processeur et mémoire), ce coût peut cependant s'avérer prohibitif. En particulier, les appels de services n'engendrant l'exécution que de quelques instructions machines (tels que ceux effectuant uniquement des opérations d'accès à des variables distantes) gagneraient à être pris en charge par un processus léger spécial préexistant sur chaque module, de façon à éviter la phase de création de processus. C'est pourquoi PM² intègre un tel mode de fonctionnement pour chacune des trois variantes des LRPC. Un LRPC effectué dans ce mode est appelé *Quick LRPC*.

Il impose cependant le respect d'une contrainte : le traitement associé à un service exécuté en mode « Quick » ne peut pas être bloquant, faute de quoi il risquerait d'entraîner des situations d'interblocage entre les modules. Étant donné qu'une application PM² se présente sous la forme d'un programme C classique et qu'elle ne nécessite pas d'autre outil que le compilateur C pour produire un exécutable, le respect de la contrainte précédente ne peut être assuré que par le programmeur.

Notons que des solutions connues à ce problème existent. La plus élégante est certainement celle proposée par Wallach et all. dans leur réalisation des *messages actifs optimistes* [162]. Dans cette approche, un *message actif* [161] est exécuté dès son arrivée sur un site par un mécanisme analogue à un déroutement causé par un signal et donc sans création de processus léger. Cependant, dès que le code associé au traitement de ce message effectue un appel bloquant, alors un processus léger est créé pour prendre en charge la suite du traitement, libérant ainsi le traitant d'interruption.

L'intégration d'un tel mécanisme dans PM² ne fait cependant pas partie de nos objectifs à court terme, car il présente un intérêt relativement faible dans le contexte d'utilisation de l'environnement, où la création de processus légers est clairement encouragée.

5.2.3 Mise en œuvre et interface de programmation

Une application PM² se compose d'un ensemble de modules placés sur les nœuds d'une architecture distribuée. Ces modules, qui représentent des « conteneurs » pour les processus légers, correspondent à des processus lourds (en l'occurrence des processus UNIX) et sont donc caractérisés par des fichiers exécutables directement issus de la compilation de fichiers sources.

Parmi l'ensemble des fonctions composant un module, certaines peuvent représenter des *services*, c'est-à-dire des points d'entrée pour les appels de procédure à distance légers. La déclaration de ces fonctions « spéciales » doit suivre une logique différente de celle d'une fonction classique, car elle doivent pouvoir être sollicitées par un processus léger se trouvant dans un module extérieur, s'exécutant potentiellement sur un nœud de nature différente.

Nous allons donc illustrer concrètement cette démarche sur un petit exemple « d'école » : la mise en place d'un service permettant le calcul du produit de toutes les valeurs d'un intervalle d'entiers donné. Plus formellement, il s'agit, étant donné deux entiers a et b (avec $a \leq b$), de calculer le nombre $a \times (a+1) \times ... \times (b-1) \times b$.

Pour bien comprendre la démarche, nous allons partir de la version classique de cette fonction et examiner, étape par étape, sa transformation de manière à la rendre accessible dans l'environnement distribué PM².

Une implantation possible en langage C de la version classique de cette fonction (que nous appellerons `produit`) est donnée ci-après (figure 5.5).

```
long produit(int inf, int sup)
{ int i;
  long prod;

   prod = 1;
   for(i=inf; i<=sup; i++)
      prod *= i;
   return prod;
}
```

FIG. 5.5 - *Fonction C calculant le produit* $inf \times (inf + 1) \times ... \times (sup - 1) \times sup$.

Pour rendre un service accessible à l'extérieur, il faut *exporter* son interface, c'est-à-dire son *nom*, le profil de ses *paramètres* et de ses *résultats*. La section suivante montre comment s'effectue typiquement cette « exportation » dans l'environnement PM^2.

5.2.3.1 Interface d'un service : nom, paramètres, résultats

Dans l'environnement PM^2, les services sont tous globaux à l'application, ce qui oblige à assigner un nom symbolique distinct à chacun d'eux. Pour assurer cette propriété, l'énumération de tous les services d'une application doit être effectuée de façon monolithique et par conséquent dans un même fichier. De façon concrète, cette énumération débute par le mot-clé `BEGIN_LRPC_LIST`, puis contient des littéraux désignant les noms des services séparés par des virgules et se termine par le mot-clé `END_LRPC_LIST`.

Cette organisation a l'avantage de permettre à l'environnement d'assigner un numéro interne unique pour chaque service. C'est ce numéro, référencé par le biais d'un littéral énumératif du langage, qui servira de *nom* pour le service. Cela permet, par rapport à un nommage réellement symbolique des services (*i.e.* chaînes de caractères) un accès direct (indicé) à différentes informations stockées dans des tables internes. L'inconvénient de cette « centralisation » des déclarations est rencontré lorsqu'il s'agit de « fusionner » plusieurs applications existantes, car il faut alors obligatoirement rassembler les différentes définitions en un seul fichier.

Une fois fixé le nom du service, il reste à préciser le profil de ses paramètres en *entrée* et en *sortie*. Dans l'exemple que nous avons choisi, la fonction `produit` nécessite *deux entiers* en entrée et renvoie *un entier long* en sortie. En les nommant respectivement `inf`, `sup` et `prod`, la déclaration du profil du service `PRODUIT` doit s'effectuer comme indiqué sur la figure 5.6. Une telle déclaration mène à la définition interne de deux types de structures de données — `LRPC_REQ(PRODUIT)` et `LRPC_RES(PRODUIT)` — capables respectivement de stocker les paramètres d'entrée et les paramètres de sortie du service `PRODUIT`. Ces types étant destinés à être utilisés à la fois par les *clients* et par les *serveurs* du service, il est commode de placer la déclaration des profils des services d'une application dans un même fichier d'en-tête, à la suite de la liste des services (figure 5.6).

Avant d'examiner les aspects concernant l'utilisation des services dans les applications, il reste un point technique à fixer : l'écriture de fonctions « souches » qui permettront au système d'assurer l'acheminement des données (paramètres et résultats) sur le réseau et leur conversion entre machines de formats différents.

```
/*                        Fichier rpc_def.h                          */
#include <pm2.h>
/* Liste de tous les services disponibles dans l'application : */
BEGIN_LRPC_LIST
   ...
   PRODUIT,
   ...
END_LRPC_LIST
...
/* Paramètres et résultats du service "PRODUIT" : */

LRPC_DECL_REQ(PRODUIT, int inf; int sup;) /* entrée */
LRPC_DECL_RES(PRODUIT, long prod;)        /* sortie */
...
```

FIG. 5.6 - *Définition de l'interface d'un service.*

5.2.3.2 Fonctions « souches »

Pour permettre à l'environnement PM² d'effectuer les opérations d'*empaquetage* et de *désempaquetage* des paramètres (resp. résultats) des LRPC, le programmeur doit fournir le corps de quatre fonctions « souches ». Le corps d'une telle fonction ne contient typiquement que des appels à des fonctions de base capables d'empaqueter (fonctions `pm2_pk*`) ou de désempaqueter (fonctions `pm2_upk*`) les types de base du langage C. La figure 5.7 illustre leur utilisation dans le cadre du service `PRODUIT`.

```
#include "rpc_defs.h"

PACK_REQ_STUB(PRODUIT)
   pm2_pkint(&arg->inf, 1, 1);
   pm2_pkint(&arg->sup, 1, 1);
END_STUB

UNPACK_REQ_STUB(PRODUIT)
   pm2_upkint(&arg->inf, 1, 1);
   pm2_upkint(&arg->sup, 1, 1);
END_STUB

PACK_RES_STUB(PRODUIT)
   pm2_pklong(&arg->prod, 1, 1);
END_STUB

UNPACK_RES_STUB(PRODUIT)
   pm2_upklong(&arg->prod, 1, 1);
END_STUB
```

FIG. 5.7 - *Fonctions souches pour le service* `PRODUIT`.

Ces fonctions sont typiquement décrites dans un fichier source séparé et sont liées aux différents modules d'une application lors de la phase d'édition de liens. Lors du déroulement

de l'application, elles sont **automatiquement** appelées — de façon transparente pour le programmeur — à chaque fois que cela est nécessaire.

La réalisation de cette dernière étape clôt la spécification de l'interface d'un service. Après avoir décrit cette interface, le programmeur peut se concentrer sur la conception de l'application elle-même, c'est-à-dire sur l'implantation des différents services au sein des différents modules.

5.2.3.3 Implantation d'un service : déclaration et code

Une application PM^2 est constituée d'un ensemble de modules, exportant chacun un certain nombre de services. Bien que la plupart des applications actuellement développées avec PM^2 adoptent un fonctionnement de type SPMD (*i.e* sont constituées d'un ensemble de modules identiques[5]), certains services peuvent ne pas être implantés dans tous les modules, ou même être implantés de façon différente suivant les modules. Cette dernière caractéristique est très utile en contexte hétérogène, où il est alors possible d'adapter l'algorithme implanté dans le service en fonction des possibilités de la machine sous-jacente. Par exemple, un service dont la vocation serait de calculer la somme de deux vecteurs ne serait pas implanté de la même façon sur une station de travail classique que sur le frontal d'une machine à architecture synchrone (SIMD).

Pour ces raisons, contrairement à la spécification de l'interface, l'implantation d'un service est une opération *locale* à chaque module. Elle consiste à fournir le code correspondant au traitement associé au service et à l'enregistrer dans le système, via l'appel de l'instruction `DECLARE_LRPC` au lancement du module. Le code correspondant à un service est défini par une suite d'instructions C classiques encapsulée par les directives `BEGIN_SERVICE(`*nom_service*`)` et `END_SERVICE(`*nom_service*`)` (figure 5.8).

Dans le corps d'un service, deux variables sont implicitement déclarées. Il s'agit d'une structure contenant les paramètres de l'appel (de type `LRPC_REQ(`*nom_service*`)` et de nom `req`) et d'une structure dans laquelle il faut ranger les résultats à retourner à l'appelant (de type `LRPC_RES(`*nom_service*`)` et de nom `res`). Ce sont pratiquement les seules différences entre un corps de service et une fonction C classique. À titre illustratif, la figure 5.8 montre le corps complet du traitement associé au service `PRODUIT` dont nous avions précédemment donné la spécification.

5.2.3.4 Appel d'un service

L'appel de procédure à distance léger constitue le principal mécanisme de décomposition parallèle des applications proposé par PM^2. Bien que l'objectif principal du projet ESPACE soit de fournir un environnement permettant une virtualisation totale des architectures, le support d'exécution PM^2, qui en constitue la couche de base, fournit des mécanismes de gestion *explicite* du parallélisme. C'est pourquoi, dans les applications développées directement au-dessus de PM^2, chaque appel de procédure à distance doit comporter la spécification d'un module cible.[6]

Nous l'avons vu, le modèle de programmation PM^2 permet l'utilisation de trois variantes du mécanisme de LRPC. Chacune de ces variantes se décline en deux « versions », selon qu'elle

5. au format d'exécutable près

6. Dans les applications utilisant des couches logicielles plus hautes, comme par exemple le régulateur LBMP, la gestion de la localisation est évidemment entièrement prise en charge par le système.

```
#include "rpc_defs.h"

BEGIN_SERVICE(PRODUIT)
  int i;

   res.prod = 1;
   for(i=req.inf; i<=req.sup; i++)
      res.prod *= i;
END_SERVICE(PRODUIT)

int main()
{
   pm2_init_rpc();
   DECLARE_LRPC(PRODUIT);
   pm2_init();

   pm2_exit();
   return 0;
}
```

FIG. 5.8 - *Implantation du service* PRODUIT.

entraîne la création d'un processus léger ou pas (*i.e.* mode *Quick*). Nous allons maintenant présenter la syntaxe d'utilisation de ces primitives, en supposant l'existence de deux variables : `request` de type `LRPC_REQ(`*nom_service*`)` et `result` de type `LRPC_RES(`*nom_service*`)` :

LRPC synchrone Un LRPC synchrone s'effectue à l'aide de l'appel à une seule primitive, qui est bloquante pour le processus appelant. La syntaxe de cette primitive est très simple dans le cas d'un appel ne déclenchant pas la création de processus :

`QUICK_LRPC(`*module*, *nom_service*, `&request, &result)`

Avant un tel appel, le programmeur doit naturellement initialiser les champs de la structure `request`. Après l'appel, les résultats sont stockés dans la variable `result`. La version déclenchant la création d'un processus comporte deux paramètres supplémentaires, qui représentent respectivement la *priorité* que l'on désire attacher au processus « serveur » et sa *taille de pile*. Nous reviendrons sur les priorités dans la section consacrée à l'ordonnancement des processus dans PM². La taille de pile, quant à elle, sera discutée dans le chapitre consacré à l'implantation de PM². Nous nous contentons de signaler ici qu'il est souvent préférable de laisser le système déterminer lui-même (en indiquant `DEFAULT_STACK`) la taille de pile adaptée au traitement du service, en fonction de la taille des arguments et des caractéristiques du processeur sous-jacent. La syntaxe est donc la suivante :

`LRPC(`*module*, *nom_service*, *priorité*, *taille_de_pile*, `&request, &result)`

LRPC à attente différée L'appel de procédure à distance à attente différée permet au processus appelant de continuer son exécution en parallèle avec le processus serveur de la requête. Nous l'avons vu, son utilisation s'effectue en deux étapes. La première consiste au déclenchement effectif du traitement et la deuxième à la récolte des résultats. Le lien entre ces deux phases réside dans l'utilisation d'une *clé* permettant, en quelque

sorte, de « désigner » un point de rendez-vous entre le *client* et le *serveur*. Si l'utilisation de cette clé est complètement transparente pour le processus serveur, elle est explicite côté client et se matérialise par une variable de type `pm2_rpc_wait`. La syntaxe d'un tel appel (avec création de processus) est illustrée en reprenant l'exemple du service `PRODUIT` (figure 5.9).

```
{ LRPC_REQ(PRODUIT) request;
  LRPC_RES(PRODUIT) result;
  pm2_rpc_wait cle;

   request.inf = 1; request.sup = 100;
   LRP_CALL(module, PRODUIT, STD_PRIO, DEFAULT_STACK,
                                &request, &result, &cle);
   ...
   LRP_WAIT(&cle);
   printf("100! = %ld\n", result.prod);
}
```

FIG. 5.9 - *Appel du service* `PRODUIT` *pour calculer la factorielle de* 100.

La syntaxe de la version sans création de processus est similaire, à ceci près qu'elle ne comporte pas les arguments concernant la priorité et la taille de pile.

LRPC asynchrone L'appel de procédure à distance asynchrone est en quelque sorte une version « dégradée » des LRPC, qui n'attend pas de résultat de la part du traitement effectué. Les primitives associées nécessitent donc moins de paramètres, ce qui donne

`ASYNC_LRPC(`*module*, *nom_service*, *priorité*, *taille_de_pile*, `&request)`

pour la version avec création de processus et

`QUICK_ASYNC_LRPC(`*module*, *nom_service*, `&request)`

pour la version sans création.

5.2.4 Discussion

Dans PM^2, le choix de *l'appel de procédure à distance léger* en tant que mécanisme principal d'expression du parallélisme résulte d'une réflexion guidée par des objectifs multiples :

Virtualisation L'environnement PM^2 est destiné au support d'applications comportant un degré de parallélisme massif à comportement parfois imprévisible. Dans ce cadre, nous avons évoqué l'intérêt de fournir aux programmeurs une virtualisation de l'architecture sous-jacente, leur permettant ainsi d'exprimer tout le parallélisme potentiel que contiennent leurs applications. L'un des fondements principaux de l'approche PM^2 réside dans l'exploitation de processus légers pour réaliser cette virtualisation. Grâce à leur très bonne adéquation au support du parallélisme fin, ceux-ci représentent un outil système efficace capable de prendre en charge un nombre conséquent de flots d'exécution asynchrones.

En contexte distribué, il ne suffit pas de juxtaposer un mécanisme de communication entre processus lourds au mécanisme des processus légers pour conserver ces bonnes propriétés. En particulier, il est nécessaire de concevoir un modèle de programmation

couplant fortement ces deux mécanismes, de façon à faire disparaître la notion de processus lourd au profit d'une description de l'application uniquement exprimée en terme de processus légers. Dans ce cadre, le choix du modèle de découpage des applications en tâches parallèles est prépondérant.

Facilité de conception L'environnement PM², contrairement à des environnements tels que Nexus qui ont clairement été conçus pour servir de cible à des compilateurs de langages parallèles, est d'abord destiné à être utilisé par des concepteurs d'applications. Par conséquent, le modèle de programmation qu'il propose et les mécanismes qui lui sont associés doivent permettre au programmeur d'exprimer très facilement le parallélisme inhérent à son application, y compris lorsque celle-ci comporte un nombre de tâches important.

Cette facilité de conception dépend non seulement de l'aptitude du modèle à favoriser une expression « naturelle » du parallélisme intra-application, mais aussi de la clarté avec laquelle cette expression peut être effectuée, au bout du compte, dans un fichier source.

Efficacité L'environnement PM² a pour finalité de permettre l'exécution d'applications parallèles sur architectures distribuées de façon la plus efficace possible. L'efficacité d'une application parallèle dépend principalement de deux caractéristiques : l'efficacité de son découpage parallèle et de la répartition des tâches résultantes sur les processeurs de l'architecture, et l'efficacité des mécanismes intrinsèques à l'environnement tels que la gestion des processus ou des communications.

Même si la première caractéristique semble relativement indépendante des choix de conception du support d'exécution, force est de constater que ces derniers ont une influence sensible sur celle-ci. En effet, une application régulée de façon optimale[7] peut très bien avoir un temps d'exécution décevant à cause d'une mauvaise gestion mémoire effectuée par le système, ou encore à cause d'un protocole de transfert des messages inutilement complexe.

Nous allons maintenant examiner dans quelle mesure le mécanisme d'*appel de procédure à distance léger* apporte des propriétés intéressantes dans chacun de ces trois domaines.

5.2.4.1 LRPC et virtualisation

Dans le domaine de la virtualisation, le modèle d'exécution des LRPC comporte deux avantages principaux sur des modèles d'exécution basés sur l'envoi de messages :

Schéma d'exécution distribué Un modèle d'exécution basé sur l'envoi de messages impose que chaque émission d'information (*i.e.* de message) soit dirigée vers une destination clairement identifiée et préexistante. De ce fait, la conception d'un programme manipulant des milliers de tâches dont les liens s'apparentent à un graphe irrégulier est souvent une entreprise complexe. De plus, comme chaque création de processus léger distant nécessite classiquement la transmission de deux messages (le premier contient la requête de création, le second contient l'identificateur de la nouvelle tâche), une stratégie

7. sur un plan théorique...

consistant à créer un nouveau processus pour chaque tâche de l'application se révélerait très inefficace dans ce cas. C'est pourquoi elle est souvent abandonnée au profit d'une organisation basée sur un nombre fixé de processus « serveurs » de calcul, c'est-à-dire une organisation *maître-esclave*. Une telle solution va à l'encontre du principe de *virtualisation* de l'architecture, en limitant artificiellement le degré de parallélisme intra-application.

L'*appel de procédure à distance léger* constitue, quant à lui, un mécanisme parfaitement adapté à un schéma de fonctionnement basé sur la création d'un processus pour chaque tâche parallèle d'une application. Il suffit pour cela d'appliquer un principe de découpage des algorithmes tel que l'algorithme *diviser pour régner*, qui génère dans ce cas une arborescence de tâches. L'implantation d'une telle méthode de découpage à l'aide d'appels de procédure à distance légers génère à l'exécution une arborescence de processus analogue à celle des tâches. Contrairement à une implantation basée sur l'envoi de messages, les relations de dépendances entre tâches (père-fils) sont dans ce cas gérées par le système et non par le programmeur, ce qui favorise la conception d'applications massivement parallèles suivant ce schéma.

Mobilité Dans un environnement basé sur l'envoi de messages, les différents processus légers d'une application ont chacun un *nom global* unique dans le système. Ce nommage global des activités d'un programme constitue la base nécessaire au fonctionnement du mécanisme d'envoi de message. Chaque fois qu'un nouveau processus est créé dans le système, un nom unique (parfois, il s'agit d'une boîte aux lettres ou d'un port de communication) lui est assigné et est renvoyé au processus « créateur ». La connaissance du nom d'un processus du système étant nécessaire à tout processus désirant lui envoyer un message, les différents processus d'une application utilisant ce mécanisme possèdent souvent, parmi leurs données locales, une ou plusieurs références (*i.e.* noms) sur d'autres processus. Pour des raisons évidentes d'efficacité, une telle référence renferme directement, de façon codée, la localisation du processus (le nœud sur lequel il s'exécute) et son identificateur local sur le nœud. Cette caractéristique rend très délicate (et forcément très inefficace) toute tentative d'implantation d'un mécanisme de migration des processus durant leur exécution.

Un modèle d'exécution basé sur l'*appel de procédure à distance léger* ne souffre pas d'un tel handicap. En effet, dans une application utilisant les LRPC, toutes les références sur un processus sont connues et gérées par le système (et uniquement par lui). De plus, comme nous le verrons dans une section ultérieure, certains processus ne sont référencés par aucun autre, ce qui autorise leur migration de façon transparente. Ces caractéristiques font que l'implantation d'un mécanisme de migration de processus dans ce contexte est une possibilité assez attrayante qui, de plus, peut être réalisée à un coût raisonnable. Ceci montre que le mécanisme des LRPC offre un modèle d'exécution très peu sensible au changement de localisation dynamique des processus, ce qui constitue une caractéristique intéressante dans l'optique d'une virtualisation réaliste de l'architecture.

5.2.4.2 LRPC et facilité de conception

De par le fait qu'ils constituent l'extension parallèle du mécanisme d'appel procédural classique, les appels de procédure à distance facilitent grandement la conception des applications

parallèles :

Découpage naturel La parallélisation d'applications suivant un modèle de type *diviser pour régner* est une démarche simple et naturelle, peut-être même la plus naturelle connue à ce jour. Il s'agit d'effectuer la découpe récursive d'un problème en sous-problèmes évaluables en parallèle, jusqu'à aboutir à la génération de problèmes élémentaires. Cette démarche donne naturellement naissance à des programmes comportant des procédures récursives, dont la cascade d'appels représente une structure arborescente.

Le mécanisme d'*appel de procédure à distance léger* est parfaitement adapté à l'expression parallèle de ces algorithmes, puisqu'il suffit de remplacer les appels de procédure séquentiels par des appels de procédure légers distants (à attente différée) pour en obtenir une version parallèle exploitable sur une architecture distribuée.

Parallélisme massif Au sein d'une application parallèle, la gestion de milliers de processus s'exécutant de façon asynchrone peut poser des problèmes lorsqu'il s'agit d'exprimer leurs multiples interactions. Avec un modèle de programmation basé sur l'envoi de message, le programmeur est obligé de gérer « manuellement » une impressionnante liste de dépendance entre processus (*qui connaît qui ?*). Puis il doit expliciter, pour chaque type de comportement, les actions à effectuer lors de la réception de messages, éventuellement en fonction de leur type. En fait, la conception de telles applications s'apparente réellement à la programmation d'un gigantesque automate distribué, chaque envoi de message pouvant déclencher la transition d'un état à un autre. Non seulement cette façon de programmer est très difficile à appréhender, mais en plus la *correction* de l'algorithme résultant est également très difficile à prouver.

Ces inconvénients disparaissent avec un modèle de programmation basé sur l'*appel de procédure à distance léger*. En effet, d'une part les dépendances entre processus sont gérées par le système et non par le programmeur, ce qui permet à ce dernier de se focaliser sur l'expression du parallélisme (*comment découper ?*) plutôt que sur la désignation des processus (*qui prend en charge quoi ?*). D'autre part, les mécanismes de LRPC favorisent fortement la conception de programmes distribués « algorithmiquement corrects », car ils représentent l'extension parallèle d'un mécanisme — l'appel de procédure — très bien maîtrisé en contexte séquentiel.

Séparation du code algorithmique du code de transfert des données Dans une application utilisant un modèle de programmation basé sur l'envoi de message, le code correspondant aux opérations de préparation des données (empaquetage, désempaquetage) avant émission ou d'extraction après réception est souvent « mêlé » au reste de l'application, c'est-à-dire au code correspondant à l'algorithme proprement dit. Par exemple, le code source des applications utilisant la bibliothèque PVM est pratiquement toujours[8] clairsemé d'instructions telles que `pvm_pk*` ou `pvm_upk*`. Non seulement cette caractéristique atténue la lisibilité du code, mais elle augmente par la même occasion les risques d'erreur en ce qui concerne le respect du protocole de transmission. En effet, les opérations d'empaquetage et de désempaquetage correspondant à un type de message donné peuvent être écrites dans des fichiers séparés, ce qui rend la vérification de leur conformité moins facile[9].

8. Nous n'avons, à ce jour, pas trouvé de contre-exemple

9. Que celui qui n'a jamais inversé l'ordre de deux appels à `pvm_pkint` me jette la première pierre !

Ce problème ne se pose pas avec les applications utilisant le mécanisme d'*appel de procédure à distance* puisque le code de transfert des données est clairement localisé au sein de fonctions souches (généralement quatre par service) elles-mêmes définies une fois pour toute dans chaque application. Avec PM^2, le programmeur est fortement encouragé à écrire ces fonctions dans un même fichier, ce qui limite fortement les risques d'incohérence entre les opérations d'empaquetage et les opérations de désempaquetage. Dans ce cadre, il pourrait être envisagé, comme dans certains systèmes basés sur un modèle client-serveur [136, 137], d'utiliser un compilateur générant automatiquement les souches à partir du profil des fonctions constituant les *services* d'une application.

5.2.4.3 LRPC et efficacité

Le domaine applicatif appréhendé dans le cadre du projet ESPACE concerne les applications de calcul scientifique, où il est question de faire face à un flot de parallélisme massif dont le comportement n'est pas prévisible, plutôt que les applications de simulation [10] dans lesquelles les entités parallèles sont clairement identifiées et où l'irrégularité concerne les interactions entre celles-ci.

Dans ces applications, des tâches parallèles sont générées à certains points de l'exécution. La plupart du temps, ces tâches sont générées « par groupes » et résultent du fractionnement d'un problème en sous-problèmes potentiellement solvables en parallèle. Lorsque l'exécution des sous-tâches correspondantes est terminée, les résultats sont communiqués à la tâche mère qui peut alors continuer son exécution.

Un environnement principalement destiné au support de ces applications doit présenter un modèle d'exécution maximisant l'efficacité de ces opérations. Le mécanisme d'*appel de procédure à distance léger* s'avère particulièrement adapté à ce cadre :

Communications sous-jacentes réduites au minimum Un modèle d'exécution basé sur l'envoi de message offre deux possibilités lorsqu'il s'agit d'exécuter une nouvelle tâche dans le système. La première consiste à envoyer la requête correspondante à un processus léger pré-existant, s'apparentant à un serveur de calcul. Nous avons évoqué les nombreux problèmes d'une telle approche en section 2.3.1. La deuxième possibilité consiste en la création d'un nouveau processus léger pour prendre en charge la tâche. Le problème est que cette méthode implique au minimum (c'est-à-dire dans le cas où est possible de transmettre les paramètres lors de la création) la transmission de trois messages entre les processus : le premier pour commander la création, le second pour retourner l'identificateur du nouveau processus et le troisième pour le retour du résultat. On le voit, la transmission du second message est inutile puisque qu'aucun message ne sera plus envoyé du processus père vers le processus fils.

Le mécanisme d'*appel de procédure à distance léger* ne possède pas cet inconvénient, car le processus léger dont il déclenche la création reste anonyme pour le processus appelant.

Prise en compte des requêtes prioritaires Comme nous le détaillerons dans la section suivante (consacrée à l'ordonnancement des processus dans PM^2), il est important, dans une application parallèle, de pouvoir assurer la prise en compte au plus tôt des requêtes « prioritaires » par rapport aux autres, de façon à favoriser par exemple l'exécution de

10. Ce qui n'empêche pas PM^2 d'être utilisé en tant que support pour de telles applications.

traitements dont on connaît à l'avance leur tendance à générer du parallélisme. Avec un modèle basé sur l'envoi de message (asynchrone), une telle propriété est délicate à assurer, car le principe de stockage des messages dans des boîtes aux lettres constitue un écueil en ce qui concerne la « réactivité » d'une application.

Le mécanisme d'*appel de procédure à distance*, couplé avec un ordonnancement préemptif des processus présente une solution adéquate à ce problème car son principe de fonctionnement garantit la création d'un processus (suivie de son exécution[11]) dès l'arrivée de la requête sur un module. La prise en compte des priorités peut alors s'effectuer en agissant sur les vitesses de progression des différents processus du module.

5.3 Concurrence

La principale caractéristique de l'environnement PM² est d'être centré sur l'utilisation de processus légers pour la prise en charge du parallélisme inhérent aux applications. Lorsqu'un *appel de procédure à distance léger* est effectué, un nouveau processus est créé[12] dans le module cible et s'ajoute à ceux qui s'exécutaient déjà dans ce dernier. Les principaux intérêts liés à cette utilisation (recouvrement des communications, efficacité des mécanismes de base) ont été largement évoqués dans les chapitres précédents, aussi n'y reviendrons-nous pas ici.

Dans cette section, nous décrivons la stratégie de gestion de la concurrence intra-modules dans PM², c'est-à-dire la stratégie d'ordonnancement appliquée dans chacun des modules d'une application. Cette stratégie consiste en un ordonnancement **préemptif avec priorité** des différents processus. Nous commençons donc par montrer les intérêts d'une politique *préemptive* par rapport à une politique *non-préemptive*, puis nous décrivons comment est effectuée la gestion des priorités des processus dans un contexte « totalement préemptif » tel que celui proposé par PM². Enfin, nous évoquons les travaux effectués au sein de notre équipe en matière d'*ordonnancement global* d'applications, c'est-à-dire d'ordonnancement de processus sur l'ensemble des nœuds d'une configuration.

5.3.1 Ordonnancement préemptif

Dans la plupart des environnements de programmation parallèle à base de processus légers existants (cf. chapitre précédent), la stratégie d'ordonnancement des processus légers peut varier d'un nœud d'une configuration à l'autre, suivant la bibliothèque de gestion des processus utilisée par l'implantation sur chaque type de machine. En conséquence, l'ordonnancement peut être préemptif sur certains nœuds et non-préemptif sur d'autres au sein d'une même application. Dans ce cas, l'ordonnancement des processus ne fait pas partie du modèle d'exécution de l'environnement et les applications ne peuvent donc en exiger aucune propriété. Hormis dans le cas particulier concernant l'utilisation en tant que cible pour des compilateurs, cette approche introduit de nombreuses limitations quant à la phase de conception des applications parallèles, puisqu'elle doit se plier à la fois aux contraintes d'un ordonnancement préemptif et d'un ordonnancement non-préemptif. Par exemple, il est nécessaire d'éviter l'exécution concurrente de portions de code non-réentrantes (à cause du risque de préemptivité), mais il n'est pas possible de garantir l'absence de famines (à cause du risque de non-préemptivité).

11. éventuellement entrelacée avec celles d'autres processus
12. Sauf indication contraire

Pour ces raisons, certains environnements, tels qu'Athapascan-0a [31], garantissent un ordonnancement *non-préemptif* des processus sur chacun des nœuds d'une configuration. Pour cela, l'éventuelle préemption disponible au niveau des diverses bibliothèques utilisées est désactivée le cas échéant. Cette approche constitue un progrès par rapport à la précédente, car elle fournit au programmeur une garantie de « comportement » de l'ordonnancement des tâches de son application. Entre autres, il n'est pas nécessaire de protéger les portions de codes à la fois non-réentrantes et non-bloquantes (cf. chapitre 3).

Cependant, dans le cadre applicatif fixé par le projet ESPACE qui entend fournir un environnement d'exécution pour une large diversité d'applications parallèles irrégulières, une approche « *non-préemptive* » s'avère trop restrictive. Au contraire, c'est une approche *préemptive* généralisée[13] qui a été retenue. Voici les principaux arguments ayant motivé ce choix :

Virtualisation Un des objectifs du projet ESPACE — la *virtualisation de l'architecture* — est de faire en sorte que ces processus légers soient véritablement perçus par le programmeur comme un ensemble de *processeurs virtuels* de cardinalité infinie. Dans cette optique, l'ordonnancement *préemptif* des processus apparaît clairement comme une nécessité :

- Le modèle d'exécution assuré par un ordonnanceur préemptif, en faisant progresser simultanément les exécutions des différents processus, est le plus proche du fonctionnement d'une machine parallèle et donc le plus **naturel**. Sur une architecture distribuée, l'ordonnancement préemptif des processus sur chacun des nœuds fournit au programmeur une vision uniforme de l'architecture, puisque les processus qu'il manipule s'exécutent toujours concurremment, indépendamment du fait qu'ils soient placés sur le même nœud ou pas.

  Cette caractéristique est très importante pour les applications comportant des processus légers « *démons* » dont le rôle est par exemple de se synchroniser périodiquement avec leurs homologues distants (il y en a généralement un sur chaque nœud) pour décider d'éventuelles actions d'ordonnancement global à entreprendre (par exemple pour équilibrer la charge). Dans ce cas, le système doit non seulement garantir un certain degré de précision lors de l'exécution d'une instruction d'attente limitée (typiquement l'instruction `sleep`) mais il doit également garantir une « bonne réactivité[14] » des mécanismes d'interaction (en l'occurrence des LRPC) entre les différents nœuds. Seul un ordonnancement préemptif peut apporter ces garanties de façon transparente et sûre.

- En effectuant une arrivée progressive dans la gamme « stations de travail personnelles », les **machines multiprocesseurs à mémoire commune** constitueront sans doute les machines de référence des architectures distribuées de demain. Sur ces machines, des bibliothèques de gestion de processus légers permettant d'exploiter un véritable parallélisme inter-processus sont souvent disponibles (ex: *Solaris Threads* [141] sur les stations Sun multiprocesseurs) et un ordonnancement strictement non-préemptif n'a pas beaucoup de sens dans un tel contexte.

  Bien évidemment, il est quand même possible d'instaurer un ordonnancement non-préemptif des processus tout en exploitant le parallélisme de la machine. Il suffit pour cela de placer un module (ou processus UNIX) par processeur physique et

13. *i.e.* sur tous les nœuds
14. grandeur bien entendu relative à l'état de charge global de la machine

d'utiliser une stratégie non-préemptive au sein de chacun des modules. Cependant, cette démarche reste très maladroite car, en considérant la machine comme une simple architecture distribuée, elle prive l'application d'une possibilité de répartition dynamique des processus sur les processeurs qui pourrait être effectuée très efficacement par un ordonnanceur « multiprocesseurs ». On le voit, sur les machines multiprocesseurs, l'ordonnancement préemptif des processus légers est incontournable. Par conséquent, toujours dans le but d'offrir une vision uniforme de l'architecture, il s'impose donc dans l'environnement PM², prétendant exploiter des architectures hétérogènes contenant des machines multiprocesseurs[15].

Efficacité Un des objectifs principaux de PM² est de permettre une exploitation *efficace* des architectures distribuées. Au premier abord, le critère d'*efficacité* associé à la notion d'*ordonnancement* peut évoquer les bonnes propriétés de l'ordonnancement non-préemptif dans le domaine de l'efficacité (cf. chapitre 3). Cependant, si les fréquents changements de contexte induits par un ordonnancement préemptif introduisent effectivement un « surcoût » par rapport à un ordonnancement non-préemptif, ce surcoût est négligeable devant une perte d'efficacité due à une mauvaise utilisation des ressources de la machine (*i.e.* mauvais équilibrage de charge menant à l'inoccupation impromptue de certains nœuds). Or nous allons voir que la stratégie d'ordonnancement des processus sur chaque nœud peut, dans le contexte du support d'applications irrégulières, influer sur les performances d'un ordonnanceur global de processus :

- Nous avons évoqué, dans l'un des points précédents, le fait qu'un ordonnancement préemptif représentait un mode naturel de fonctionnement en contexte parallèle et qu'il garantissait une certaine « réactivité » des différents nœuds de l'architecture. En fait, cette propriété est primordiale lorsqu'il s'agit d'implanter un ordonnanceur global (par exemple un régulateur de charge) pour une application fortement irrégulière. Dans un tel contexte, où la durée de vie des différents processus légers n'est pas prévisible et encore moins « uniforme », les mécanismes chargés de « rétablir » la charge en cas de fort déséquilibre se doivent d'agir rapidement, faute de quoi des processeurs risquent d'être sous-utilisés. Avec un ordonnancement non-préemptif, il n'est pas possible de borner le temps de réaction d'un nœud vis-à-vis d'une sollicitation extérieure (en l'occurrence un LRPC).
- Certaines applications parallèles peuvent distinguer plusieurs niveaux de *priorités* parmi les différents traitements qui la composent. Par exemple, les actions d'ordonnancement dont nous parlions au point précédent se voient souvent attribuer une priorité beaucoup plus grande que les traitements classiques dans les applications parallèles, car de leur rapidité d'exécution dépendent les performances globales d'une application. Nous reviendrons plus en détail sur les priorités des processus dans la suite de cette section. Nous allons nous intéresser ici à un exemple précis illustrant la nécessité d'un ordonnancement préemptif pour un « respect efficace » des priorités en contexte distribué.

  Considérons une application où, à certains moments, il est possible de prédire que l'exécution d'une tâche mènera très sûrement à la génération de plusieurs

15. Nous verrons au chapitre suivant que l'environnement PM² n'a pas encore été porté sur de telles architectures.

sous-tâches à exécuter en parallèle. Une telle information est très intéressante à exploiter, car dans un environnement où le principal souci consiste à équilibrer la charge parmi les nœuds, l'exécution des traitements générateurs de parallélisme doit être favorisée (plus il y a de tâches « prêtes », plus il est facile de réguler la charge). Pour cela, une grande priorité peut leur être attachée. Cependant, en contexte distribué, cela ne suffit pas. En effet, si la requête de création de processus destinée à prendre en charge un tel traitement arrive sur un nœud où l'ordonnancement local est effectué de manière non-préemptive, alors elle ne sera pas prise en compte avant l'interruption (terminaison ou blocage) du processus en cours (figure 5.10). Au contraire, un ordonnancement préemptif assurerait une prise en charge quasi-immédiate[16] du traitement prioritaire, indépendamment de la durée du processus en cours d'exécution.

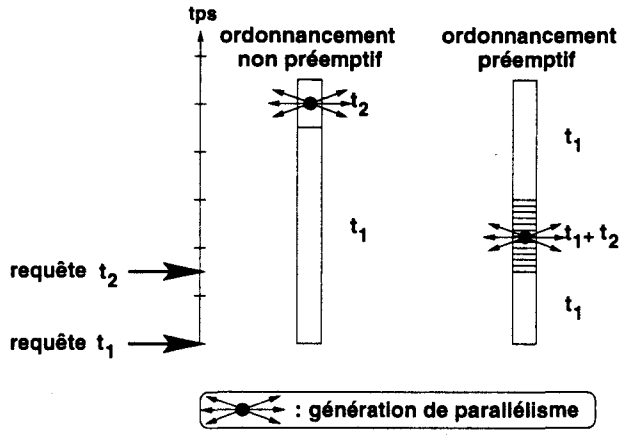

FIG. 5.10 - *Intérêt de la préemption pour la prise en compte de requêtes « prioritaires ». Ici, la tâche t_2, fortement génératrice de parallélisme, est supposée posséder une priorité plus grande (environ le double) que la tâche t_1.*

- Dans le point précédent, nous avons évoqué l'intérêt d'exploiter au plus tôt les traitements générateurs de parallélisme, pour augmenter le degré de parallélisme potentiel disponible à un instant donné. Cet avis n'est pas partagé par tous les concepteurs d'environnements parallèles. En particulier, les concepteurs de DTS [20] limitent le nombre d'activités concurrentes par nœud à un maximum de 5 (grandeur déterminée expérimentalement !). Cette limitation a pour but d'éviter des déséquilibres de la charge trop importants lorsqu'il n'y a plus d'activités à placer. En fait, ce raisonnement est typiquement lié à l'absence de fonctionnalités de migration de processus dans DTS. Dans PM^2, qui dispose de telles fonctionnalités, la stratégie est complètement opposée à celle de DTS : plus le degré de parallélisme potentiel est grand et plus il est « facile » d'équilibrer la charge dynamiquement (en utilisant la migration de processus en contexte distribué, ou de manière directe sur architecture à mémoire commune), c'est pourquoi il est important de le favoriser.

  Dans cet esprit, un ordonnancement préemptif des processus est un moyen de **pré-**

16. Plus exactement « bornée » par une constante (quantum de préemption) multipliée par le nombre de processus légers actifs sur le nœud.

server « le plus longtemps possible » ce **degré de parallélisme**. En effet, alors qu'un ordonnancement non-préemptif présente une tendance à exécuter les processus un par un, l'ordonnancement préemptif, quant à lui, assure une progression *simultanée* des différents processus et maintient donc un degré de parallélisme plus important. Pour illustrer l'importance de ce maintien du degré de parallélisme, nous allons examiner un petit exemple simple (volontairement « extrême ») montrant combien l'équilibrage de la charge d'une machine peut s'en trouver amélioré. Cet exemple consiste en une machine comportant 10 processeurs (P_1 à P_{10}, supposés identiques) sur lesquels il s'agit d'exécuter 15 processus (t_1 à t_{15}) tous créés à l'instant $t = 0$. L'ordonnanceur global de la machine ne connaît pas à l'avance la durée d'exécution de ces processus, que nous supposerons égale à 2 unités de temps pour chacun des 15 processus. Dans le cas d'un ordonnancement non-préemptif, 10 processus (par exemple t_1 à t_{10}) sont d'abord exécutés chacun des 10 processeurs, puis, au temps $t = 2$, les 5 processus restants sont exécutés chacun sur un processeur (figure 5.11, en haut).

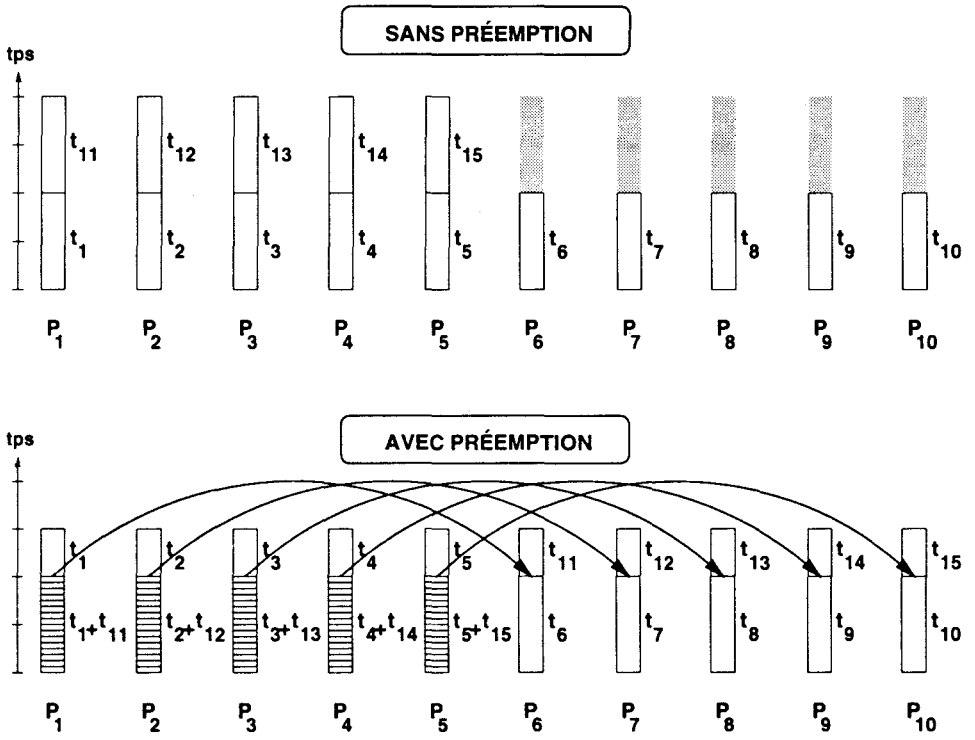

FIG. 5.11 - *Intérêt de la préemption pour l'exploitation efficace de l'architecture.*

Au contraire, avec un ordonnancement préemptif, l'exécution de tous les processus est démarrée au temps $t = 0$ (figure 5.11, en bas). À ce moment, 5 processeurs doivent supporter l'exécution concurrente de deux processus (par exemple, sur la figure, P_3 supporte l'exécution de t_3 et t_{13}). Au temps $t = 2$, la moitié des processeurs deviennent inactifs (ceux qui ne supportaient l'exécution que d'un seul processus, *i.e.* P_6 à P_10 sur la figure), alors qu'il reste 2 processus sur chacun des 5 autres processeurs. Par conséquent, l'ordonnanceur global peut *déplacer* l'exécu-

tion d'un processus depuis chacun de ces 5 processeurs vers les processeurs libres, pour équilibrer la charge. Étant donné que ces processus ont tous été exécutés deux par deux de manière entrelacée (supposée équitable), ils ont été exécutés chacun pendant une unité de temps et donc il leur reste encore une unité de temps d'exécution. On le voit, le temps d'exécution global de l'application avec un ordonnancement préemptif est donc de 3 unités de temps, contre 4 dans le cas d'un ordonnancement non-préemptif.

Notons pour terminer que l'intérêt de la préemption dans ce cas **est lié** à l'utilisation d'un mécanisme permettant de déplacer les processus d'un processeur à un autre pendant leur exécution. Entre deux processeurs appartenant à une architecture à mémoire commune, ce déplacement peut être effectué par une simple action interne à l'ordonnanceur de la machine. En revanche, entre deux processeurs ne partageant pas de mémoire physique, ce déplacement ne peut être effectué que par un mécanisme de *migration* de processus.

Pour toutes ces raisons, l'environnement PM^2 adopte un ordonnancement *préemptif* des processus légers au sein de chacun des modules d'une configuration. Le temps processeur est donc découpé en unités appelées des *quanta* (cf. chapitre 3) dont la durée réelle est paramétrable par l'application à l'aide de la primitive `pthread_settimeslice_np(`*durée*`)`. L'attribution de ces *quanta* de temps aux processus dépend de leur *priorité*, comme nous allons le voir dans la section suivante.

5.3.2 Priorités

Dans le chapitre 3, nous avons vu qu'il était intéressant pour une application multiprogrammée de pouvoir attacher différentes priorités aux processus légers qui la composent, de manière à pouvoir exprimer la notion de traitement « urgent » au système. Dans un contexte distribué tel qu'une configuration de modules PM^2, cette propriété devient une nécessité lorsqu'il s'agit d'implanter des mécanismes tels que des régulateurs de charge, comme il a été évoqué dans les sections précédentes.

Cependant, il serait réducteur de considérer les priorités comme un moyen de distinguer les traitements « normaux » des traitements « urgents » dans une application. En réalité, la notion de traitement urgent n'existe pas dans PM^2. Au contraire, nous y reviendrons un peu plus loin, il existe une *échelle de priorités* (actuellement de 1 à 100) destinée à permettre au programmeur d'exprimer différentes *vitesses d'exécution relatives* au sein de son application.

L'utilisation de différents niveaux de priorités revêt une importance particulière dans les applications dont l'exécution peut être guidée par des *heuristiques*. Typiquement, c'est le cas dans les applications d'optimisation combinatoire, où l'exploration des espaces de recherche est souvent dirigée par des priorités calculées à l'aide d'heuristiques (cf. section 2.2.5). Dans ce cas, l'exécution privilégiée des tâches les plus prioritaires d'une telle application conduit souvent[17] à une convergence plus rapide vers la solution du problème et donc à une efficacité plus importante.

Dans un environnement parallèle (en l'occurrence PM^2), il est donc intéressant de pouvoir attacher des priorités aux différents processus légers d'une telle application, de façon à pouvoir indiquer au système (en quelque sorte) le « bon ordonnancement » à effectuer pour aboutir à une convergence rapide de l'algorithme vers la solution recherchée. Du point de vue du

17. si l'heuristique est bonne !

programmeur, l'intérêt est énorme, puisqu'il n'a pas à gérer lui-même de structures de données où il stockerait les tâches en attente d'exécution, en fonction de leur priorité. Au contraire, il peut uniquement se contenter de créer les tâches de l'application, en leur assignant une priorité d'exécution et en laissant le système s'occuper du reste[18].

Dans l'environnement PM², les processus se voient donc attribuer des priorités fixées par l'application. Contrairement à ce qui se passe dans certains systèmes d'exploitation en ce qui concerne les processus lourds, les priorités ne sont pas modifiées par l'environnement et restent donc entièrement sous le contrôle du programmeur. Ces priorités sont exprimées sous la forme de nombres entiers compris entre 1 et `MAX_PRIO` (actuellement 100). Nous avons vu dans la section consacrée aux LRPC que la priorité d'un processus est spécifiée par son processus « père » au moment de l'appel. Cette priorité peut également être changée dynamiquement grâce à la primitive `pthread_setprio(`*processus*, *priorité*`)`. Il nous reste maintenant à examiner la sémantique de ces priorités dans l'environnement PM², c'est-à-dire le fonctionnement de l'ordonnanceur des processus en présence de priorités.

La première chose à noter est que la priorité d'un processus léger n'a qu'une portée *locale* au module dans lequel il s'exécute. Autrement dit, chaque module possède son propre ordonnanceur de processus légers et celui-ci ne se synchronise en aucune façon avec les autres (figure 5.12). Ce choix a été effectué en raison de la trop grande spécificité d'un ordonnanceur « *global* » qui assurerait une corrélation entre les ordonnanceurs locaux des modules. La conception d'un tel *ordonnanceur global* est néanmoins possible à l'aide du support d'exécution PM², comme nous le verrons dans la section suivante.

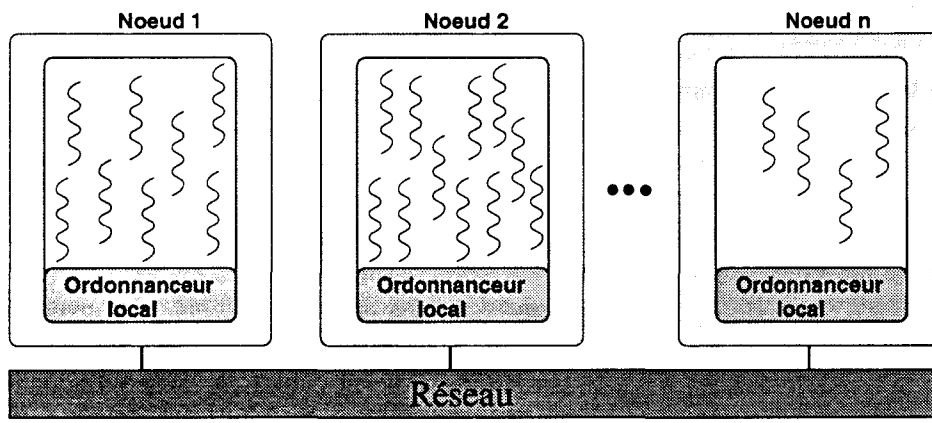

FIG. 5.12 - *Une configuration PM² consiste en un ensemble de modules contenant chacun un ordonnanceur local. En utilisant les fonctionnalités de placement et de migration de processus, il est possible d'implanter un ordonnanceur global adapté à une classe d'applications particulière.*

Dans le chapitre 3, nous avons vu que la stratégie d'ordonnancement préemptive avec priorité la plus couramment utilisée était celle du *tourniquet multiple* (section 3.4.2.2). Cependant, dans sa version de base, cette stratégie n'effectue de commutations de contexte qu'entre les processus de plus haute priorité. Compte tenu des attentes que nous avons exprimées à propos de l'ordonnancement préemptif, cette caractéristique n'est pas satisfaisante puisqu'elle va à l'encontre de la plupart des propriétés énoncées dans la section précédente. C'est pourquoi

18. Nous verrons un peu plus loin que, par contre, cela ne facilite pas le travail du système...

nous avons défini une stratégie d'ordonnancement spécifique, qui assure à la fois un respect des priorités des processus et une préemption totale entre ces derniers. Nous considérons dans la suite que les nœuds de l'architecture sont constitués de machines monoprocesseur.

Cette stratégie est définie par la propriété suivante : quels que soient x et x', deux processus légers actifs appartenant au même module (de priorités respectives $prio(x)$ et $prio(x')$), le processus x s'exécutera $prio(x)/prio(x')$ fois plus vite que le processus x' sur une période de temps suffisamment longue. Autrement dit, les priorités des processus représentent des « *vitesses d'exécution* » relatives aux autres processus. Dans l'environnement PM^2, la vitesse absolue $v(x)$ (avec $0 < v(x) \leq 1$) d'un processus x est définie ainsi :

$$v(x) = \frac{prio(x)}{\sum_{y \in \{\text{processus actifs du même module}\}} prio(y)}$$

On vérifie bien la propriété énoncée précédemment pour deux processus du même module :

$$\forall x, x' \in \{\text{processus actifs}\}, \frac{v(x)}{v(x')} = \frac{prio(x)}{prio(x')}$$

D'un point de vue pratique, cette définition est appliquée dans chacun des modules par l'ordonnanceur local en assurant[19] que, pendant un temps égal à la durée d'un quantum multipliée par la somme des priorités des processus actifs, chaque processus obtient le processeur pendant un temps équivalent à la durée d'un quantum multipliée par sa priorité.

5.3.3 Vers un ordonnancement global

La stratégie d'ordonnancement décrite dans la section précédente représente le mode de fonctionnement « brut » de l'environnement PM^2 et consiste en un ensemble d'ordonnanceurs locaux fonctionnant de façon autonome. Nous l'avons vu, ces ordonnanceurs possèdent un certain nombre de propriétés améliorant la conception d'applications parallèles. Cependant, cette conception ne peut reposer sur ces seules propriétés pour espérer obtenir une exécution efficace sur une architecture distribuée. En particulier, ces applications nécessitent l'intervention d'un mécanisme de régulation de charge capable non seulement d'effectuer un placement judicieux des processus parmi les modules, mais aussi de corriger les éventuels déséquilibres pouvant survenir dynamiquement en déplaçant les processus entre les modules[20].

Un tel mécanisme, aussi appelé « *ordonnanceur global* » d'applications, est forcément spécifique à une classe d'applications donnée, puisqu'il existe un très grand nombre de critères suivant lesquels la régulation évoquée précédemment peut être orientée. Un des objectifs du support d'exécution PM^2 est de faciliter le développement de tels régulateurs. Dans le cadre du projet ESPACE, la conception d'un régulateur spécialisé pour les applications nécessitant la « globalisation » de la notion de priorité est actuellement à l'étude et constitue le travail de thèse d'Yves Denneulin [52, 54]. Ce régulateur, nommé LBMP (*Load Balancing with Migration directed by Priorities*), a pour objectif de fixer une portée *globale*[21] aux priorités des processus

19. Ou plutôt « en faisant de son mieux pour assurer », car les créations, blocages et autres terminaisons de processus apportent évidemment des perturbations temporaires à ce fonctionnement.

20. Les expressions « charge », « placement judicieux » et « déséquilibres » sont à prendre à un sens très général.

21. c'est-à-dire de rendre la propriété des « vitesses relatives » valable même entre processus de modules différents

d'une application. Le principe appliqué par ce régulateur est de maintenir une « *quantité de priorités* » sensiblement égale dans chacun des modules d'une application, en utilisant des mécanismes de *migration* de processus légers.

5.4 La migration de processus légers

Dans l'infrastructure logicielle proposée par le projet ESPACE, le support d'exécution PM² occupe une position centrale, puisqu'il représente le noyau commun sur lequel reposent, le plus souvent indirectement, toutes les réalisations effectuées dans le cadre du projet lui-même, ou dans le cadre de collaborations inter-laboratoires. Ces réalisations concernent le très vaste domaine des applications parallèles irrégulières massivement parallèles. L'objectif du projet ESPACE est de proposer, à travers ces réalisations, un environnement logiciel permettant d'assurer l'exécution efficace de ces applications sur architectures distribuées. Comme nous l'avons déjà évoqué (cf. présentation du projet STRATAGÈME, section 2.2.1), la régulation de charge revêt une importance particulière dans ce cadre, c'est pourquoi des travaux allant dans ce sens sont menés au sein de notre équipe de recherche [76].

La diversité des applications irrégulières étudiées dans le cadre du projet STRATAGÈME indique qu'il semble peu réaliste de chercher à proposer une stratégie « universelle » de régulation de charge, laquelle serait valable quelle que soit l'application envisagée. Au contraire, il est plus judicieux d'essayer de classifier les applications irrégulières en fonction de critères communs précis et de proposer des stratégies de régulations pour chacune de ces classes d'applications. Le projet STRATAGÈME, par exemple, a abouti sur la proposition d'une classification des applications irrégulières basée sur la « prédictibilité » du graphe de précédence des tâches qui les composent [75]. Cette proposition, qui repose sur la distinction de trois grandes catégories d'applications (graphe prévisible, semi-prévisible ou imprévisible), permet, après une phase d'expertise, de déterminer le type de stratégie de régulation le mieux adapté à une application.

Malgré l'avancée que représente l'existence d'une telle classification, force est de constater qu'au sein de chacune des trois catégories se trouvent des applications de nature parfois très différentes, qu'il convient souvent de réguler de manière spécifique. Par exemple, dans la catégorie « *applications à graphe de précédence imprévisible* » se trouvent aussi bien des applications de simulation de particules que des applications d'optimisation combinatoire basées sur un algorithme *Branch & Bound*. Pourtant, au contraire des applications de simulation, les exécutions des applications d'optimisation combinatoire sont basées sur des explorations d'espaces de recherche qui peuvent être guidées par des heuristiques, ce qui constitue un facteur d'accélération important [47].

Pour ces raisons, le projet ESPACE entend fournir non seulement un ensemble de régulateurs de charge adaptés aux applications irrégulières les plus répandues, mais aussi et surtout une base générique destinée à permettre aux utilisateurs de l'environnement PM² de développer **aisément** de nouveaux régulateurs adaptés à leurs besoins spécifiques donc plus **performants**. C'est pourquoi il a été choisi de n'implanter aucune stratégie de régulation dans le noyau de l'environnement — le support d'exécution PM² — mais plutôt de les implanter sous forme de bibliothèques qu'un utilisateur peut librement incorporer à ses applications s'il le souhaite.

L'efficacité de ces stratégies de régulation, quelles qu'elles soient, dépend en partie de l'étendue des fonctionnalités offertes par le support d'exécution PM² et de l'efficacité des

mécanismes impliqués. En particulier, l'implantation de politiques de régulation capables d'équilibrer la charge d'une application en d'autres circonstances qu'à l'occasion de créations de processus nécessite d'autres fonctionnalités que l'*appel de procédure à distance léger*. Le problème initial était donc de trouver l'ensemble minimal des fonctionnalités nécessaires à l'implantation d'une vaste étendue de régulateurs de charge tout en préservant une simplicité d'utilisation maximale.

Plusieurs questions se sont alors posées lors de la conception du support d'exécution :

- Un mécanisme de migration de processus peut-il être utile pour le support des applications irrégulières?
- Si oui, dans quelle mesure est-il possible de l'intégrer dans PM^2 sans perturber le fonctionnement des autres fonctionnalités?
- La migration de processus lourds est souvent considérée comme un mécanisme dont les performances intrinsèques sont trop faibles. Est-ce aussi le cas pour la migration de processus légers?

Les sections à venir sont destinées à apporter une réponse à chacune de ces questions[22]. Dans un premier temps, nous examinerons donc la motivation qui nous a menés à la conception d'un tel mécanisme pour l'environnement PM^2. Ensuite, nous détaillerons son principe de fonctionnement et l'illustrerons sur un exemple concret. Nous poursuivrons cette étude en indiquant les limites du mécanisme et en expliquant sa cohabitation avec le mécanisme d'*appel de procédure à distance léger*. Enfin, nous terminerons par la comparaison de notre approche avec des travaux similaires.

5.4.1 Motivations

L'immense majorité des environnements de programmation parallèle et distribuée actuels, qu'ils soient basés sur un parallélisme à grain fin ou pas, ne fournissent pas de fonctionnalités de migration de processus. Il y a plusieurs raisons à cela :

Portabilité La plupart des concepteurs d'environnements parallèles revendiquent l'excellente portabilité de l'implantation qu'ils fournissent, grâce à l'utilisation de bibliothèques « standard » de gestion de processus (POSIX) ou de gestion des communications (MPI). L'avantage d'utiliser ces bibliothèques se transforme en inconvénient lorsqu'il s'agit d'implanter des fonctionnalités « hors normes », telles que la migration de processus par exemple. En effet, cette fonctionnalité, à moins d'utiliser des techniques de « points de reprise » des processus (ce qui limite fortement son domaine d'utilisation), nécessite d'être implantée au même niveau que la gestion des processus, puisqu'elle doit accéder à la représentation interne de ces derniers. Cette propriété s'applique surtout aux processus légers, dont le contexte d'exécution n'est pas « translatable[23] » sans modifications.

C'est pourquoi il n'est généralement pas possible d'implanter des mécanismes de migration sans avoir la **maîtrise** des mécanismes de gestion de processus, c'est-à-dire l'accès à la représentation interne de ces derniers. Dans le chapitre consacré à l'implantation de PM^2, nous montrons comment nous avons implanté de tels mécanismes (dans une

22. Ce qui semble d'ores et déjà indiquer que la réponse à la première question est oui...
23. relogeable d'un emplacement mémoire à un autre

bibliothèque de processus légers nommée MARCEL) tout en conservant un degré de portabilité élevé.

Efficacité La migration a une mauvaise réputation, imputable en majeure partie aux faibles performances qu'elle exhibe sur certains systèmes. En fait, il s'agit de la migration de processus lourds [25], laquelle implique parfois le transfert de plusieurs méga-octets de données (représentant le contexte du processus) d'une machine à une autre. Par conséquent, il n'est pas rare de constater, sur un réseau de stations de travail connectées par réseau éthernet, des temps de transfert de plusieurs secondes pour la migration d'un seul processus.

Si cet argument se révèle dissuasif dans un contexte appliqué aux processus lourds, il n'est pas pertinent dans un contexte appliqué aux processus légers, car les performances de la migration dans ce dernier cas sont nettement supérieures. Dans le chapitre consacré aux performances de l'environnement PM², nous montrons que le coût d'une migration de processus léger représente typiquement quelques millisecondes (de l'ordre d'une milliseconde sur la machine IBM SP2 de Grenoble, avec la bibliothèque de processus MARCEL et la bibliothèque de communication PVMe).

Difficulté d'intégration L'intégration de mécanismes de migration de processus dans un environnement de programmation parallèle n'est pas une opération triviale. En dehors des difficultés techniques rencontrées lors de son implantation, il faut surtout établir une sémantique définissant les règles d'utilisation conjointe de la migration avec les autres fonctionnalités de l'environnement. Idéalement, cette sémantique ne doit pas perturber le fonctionnement « normal » des autres mécanismes. Par exemple, dans un environnement de programmation basé sur l'envoi de message, il faut assurer que l'envoi d'un message à un processus dont on possède l'identifiant a toujours lieu correctement, même si ce dernier a été migré (*i.e.* a changé de nœud) entre temps. Pour parvenir à un tel fonctionnement, il est nécessaire de *virtualiser* les identifiants de processus, c'est-à-dire de rendre leur utilisation transparente vis-à-vis de la localisation des processus qu'ils référencent. Cette approche, adoptée par des systèmes tels que MPVM [25], a cependant de lourdes répercussions sur l'efficacité des mécanismes d'envoi de message eux-mêmes, dont l'implantation repose alors sur des mécanismes d'indirection assez coûteux.

Dans les sections à venir, nous montrons comment la migration de processus légers a été intégrée dans PM² de manière efficace et la façon dont ce mécanisme cohabite avec l'*appel de procédure à distance léger*.

Utilité L'intégration de la migration dans un environnement de programmation parallèle est une démarche guidée par un besoin profond issu des applications supportées. Par exemple, les environnements spécialisés dans le support des applications de simulations discrètes utilisent cette fonctionnalité pour « rapprocher » (*i.e.* placer sur le même site) dynamiquement les entités manifestant des affinités prononcées (*i.e.* une intense activité de communication). D'autres environnements, tels qu'Ariadne [115], l'utilisent pour résoudre des problèmes de localité de données lors de l'exécution de certains traitements. Les processus sont alors migrés là où se trouvent les données qu'ils manipulent. Enfin, des environnements tels que MPVM [25] ou UPVM [26] utilisent la migration de processus pour équilibrer la charge des nœuds de l'architecture pendant l'exécution d'une application. Cependant, pour une grande partie des environnements existants, l'intérêt

d'intégrer un mécanisme de migration de processus semble trop faible pour que cette action soit effectivement entreprise.

C'est pourquoi il n'est pas possible, sans étudier précisément la classe des applications visées, d'évaluer l'intérêt de la migration de processus dans un environnement de programmation parallèle.

Nous allons maintenant exposer notre réponse à ce dernier point, en montrant qu'un régulateur de charge disposant de la possibilité de *migrer* des processus durant leur exécution peut conduire à l'exécution plus efficace d'applications parallèles irrégulières, par rapport à un régulateur ne pouvant effectuer que du *placement* de processus.

5.4.1.1 Ordonnancement sans migration *vs* ordonnancement avec migration

Pour bien montrer l'intérêt de fournir un mécanisme de migration de processus légers dans le support d'exécution PM^2, nous allons examiner les efficacités comparées d'un ordonnanceur[24] ne disposant pas de la migration avec celle d'un ordonnanceur disposant de la migration sur un « cas d'école ». Nous supposerons, pour simplifier, que les coûts des différents transferts d'information entre les nœuds (y compris les coûts des migrations de processus) sont suffisamment petits, en regard de la durée des traitements, pour être négligés.

La figure 5.13 montre (à gauche) le graphe de précédence des tâches d'un exemple d'application parallèle irrégulière. On le voit, la durée minimale théorique de cette application sur un nombre de processeurs infini[25] est de 9 unités de temps.

Nous allons nous intéresser à l'ordonnancement de cette application sur deux processeurs, en évaluant respectivement l'efficacité d'un ordonnanceur sans migration dans ce contexte, puis celle d'un ordonnanceur avec migration.

Dans un premier temps, nous considérons un ordonnanceur ne disposant pas de la possibilité de *déplacer* une tâche durant son exécution. Par conséquent, une fois le *placement* d'une telle tâche effectué, celui-ci ne peut plus être remis en cause.

Si l'on essaie, en respectant les relations de précédence entre les tâches, toutes les façons possibles de les placer sur deux processeurs, on s'aperçoit qu'il n'est pas possible de parvenir à une durée d'exécution totale de moins de 12 unités. Cela signifie que, pour cet exemple, quand bien même l'ordonnanceur **connaîtrait à l'avance** le déroulement précis de l'application (graphe de précédence **et** durée d'exécution de chacune des tâches), il ne pourrait pas trouver de meilleur ordonnancement que celui illustré sur la figure 5.13 (à droite).

Notons que, dans cet exemple, une éventuelle possibilité d'entrelacer les exécutions des tâches sur chaque processeur (*i.e.* ordonnancement local préemptif) ne changerait rien aux choses.

Nous nous plaçons maintenant dans le cadre d'un ordonnanceur ayant la possibilité de déplacer les tâches (d'un processeur à un autre) pendant leur exécution.

En se plaçant dans le contexte « idéal » évoqué précédemment, c'est-à-dire un contexte où le déroulement précis de l'application est connu à l'avance, alors il est possible pour un ordonnanceur disposant du mécanisme de migration de trouver l'ordonnancement des tâches optimal compte tenu du nombre de processeurs disponibles. C'est ce qui est illustré dans la partie gauche de la figure 5.14. Nous n'apportons pas la preuve de cette propriété ici, car elle

24. Ici, le terme « ordonnanceur » doit être pris au sens global (régulateur) et non au sens local tel que nous l'avons évoqué jusqu'à présent.

25. En l'occurrence, trois processeurs suffiraient

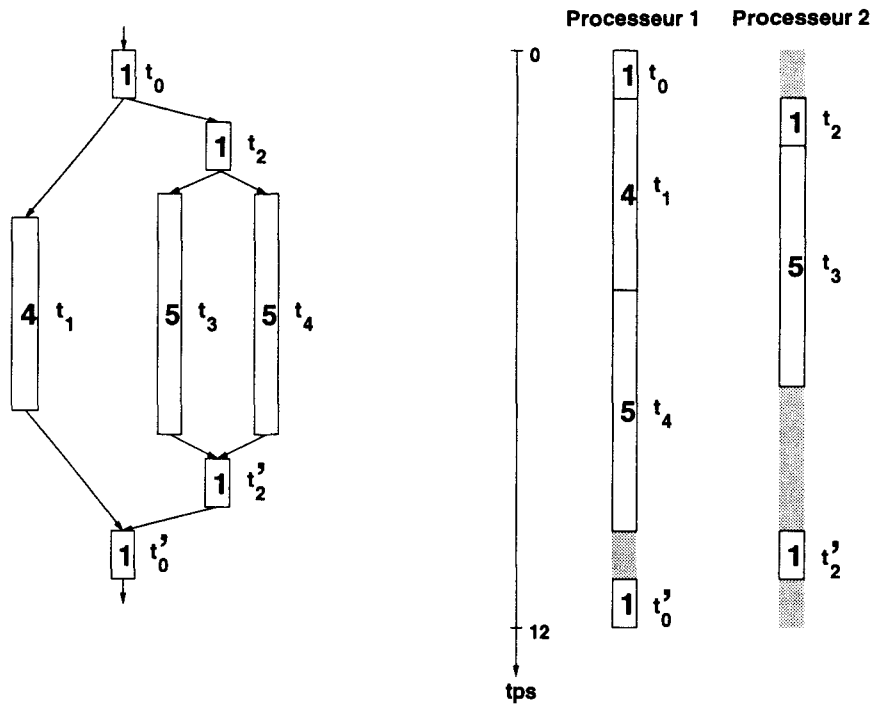

FIG. 5.13 - *À gauche : graphe de précédence des tâches d'un exemple d'application parallèle irrégulière. Les nombres inscrits à l'intérieur représentent les durées d'exécution des tâches. À droite : exemple de placement des tâches sur une architecture à deux processeurs. En fait, il n'est pas possible de faire mieux (*i.e. *obtenir une durée d'exécution totale inférieure à 12 unités de temps).*

n'aurait aucun intérêt pratique. En effet, dans les applications irrégulières « concrètes », la durée des tâches est rarement connue à l'avance. Tout au plus est-il possible de déterminer, par des méthodes de pré-calcul, le graphe de précédence de certaines d'entre elles (*applications à graphe de précédence prévisible*). C'est pourquoi nous ne poursuivrons pas plus avant l'étude de l'ordonnancement avec migration sous cette hypothèse.

Nous nous plaçons donc maintenant dans le contexte réaliste consistant à ne pas pouvoir prédire la durée d'exécution de chacune des tâches de l'application. De plus, nous allons considérer que le graphe de précédence n'est pas prévisible non plus et allons montrer qu'il est possible, à l'aide d'un simple ordonnanceur utilisant la migration pour équilibrer la charge, d'obtenir une exécution plus efficace qu'avec un ordonnanceur sans migration.

Pour cela, nous utiliserons un ordonnanceur ayant les caractéristiques suivantes :

- lors de la création d'une nouvelle tâche, celle-ci est placée sur un processeur libre si c'est possible, ou sur un processeur choisi au hasard sinon ;
- lorsque plusieurs tâches se trouvent sur le même processeur, leur exécution est entrelacée de façon équitable (ordonnancement local préemptif sans priorité) ;
- lorsqu'un processeur devient libre et qu'un autre processeur contient au moins deux tâches, alors l'une de ces tâches (au hasard) est migrée sur le processeur libre.

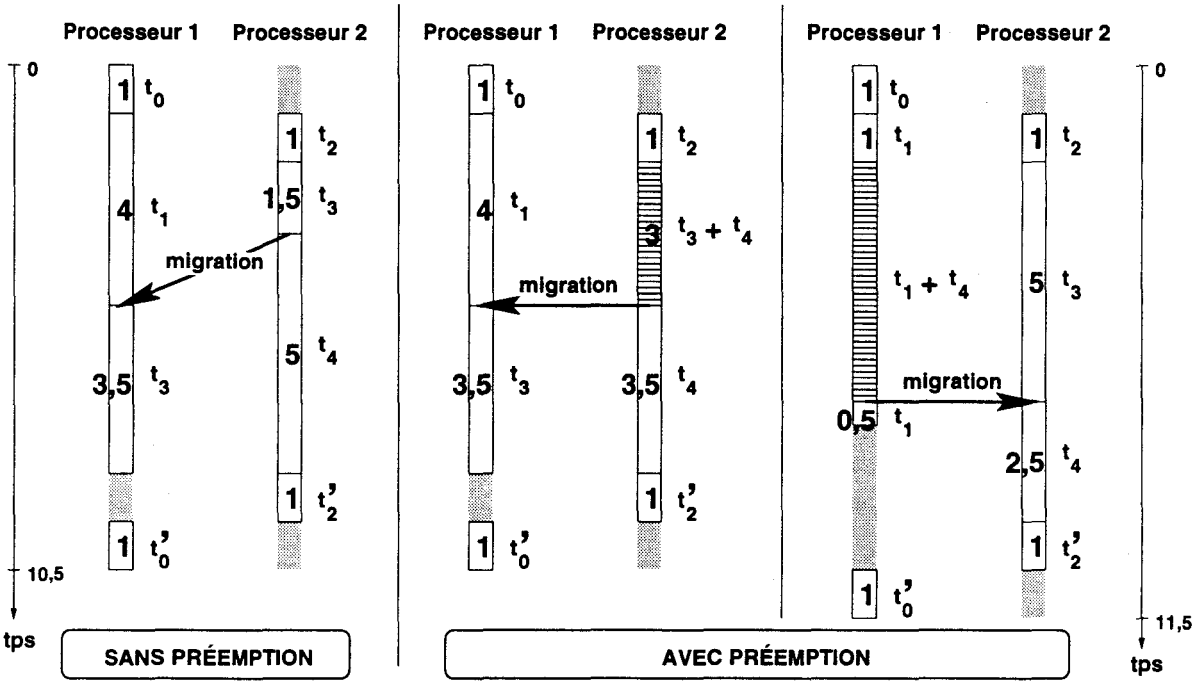

FIG. 5.14 - *Avec l'aide d'un mécanisme de migration, il est possible d'exploiter plus efficacement l'architecture distribuée. À gauche : en connaissant le graphe de précédence des tâches ET leur durée d'exécution (ce qui n'arrive pas souvent !), la migration permet d'exploiter au mieux l'architecture. À droite : dans un cas réaliste, c'est-à-dire lorsqu'on ne connaît pas la durée d'exécution des tâches, la migration couplée à un ordonnanceur préemptif permet souvent d'améliorer l'utilisation des processeurs.*

Pour simplifier le problème, nous posons également les trois hypothèses suivantes :

1. La fréquence de préemption est suffisamment grande pour rendre négligeable (en regard de la durée des tâches) la durée d'un quantum de temps élémentaire. Notons que cette hypothèse est parfaitement réaliste en pratique.

2. Le temps nécessaire à l'ordonnanceur pour détecter qu'un processeur est libre est également négligeable par rapport à la durée des tâches. Là encore, cette hypothèse est réaliste (dans l'environnement PM^2, il est possible de définir un traitement qui sera automatiquement déclenché dès qu'une variation de la charge locale d'un processeur sera détectée).

3. Le temps nécessaire à l'ordonnanceur pour trouver un processus « candidat à la migration », ajouté au coût de la migration, est négligeable. Cette hypothèse est un peu forte, surtout si le nombre de processeurs devient très grand... Néanmoins, elle ne fausse pas la comparaison avec l'ordonnanceur sans migration, pour lequel nous avons implicitement fait une hypothèse du même ordre.

L'application de la stratégie d'ordonnancement que nous venons de définir sur le graphe de précédence de la figure 5.13 mène, en raison de l'indéterminisme de l'algorithme, à plusieurs

exécutions possibles. Néanmoins, ces exécutions sont classifiables en deux catégories. Nous montrons, sur la figure 5.14 (au centre et à droite), un « échantillon » de chacune de ces deux catégories. Sur cette figure, il est facile de voir que l'exécution de l'application peut durer, selon le cas, 10,5 ou 11,5 unités de temps, ce qui constitue une exécution plus efficace que dans le cas d'un ordonnancement sans migration.

Nous venons donc de montrer, sur un cas pourtant très simple, qu'un ordonnanceur global (autrement dit un régulateur de charge) pouvait, grâce à la migration de processus, diminuer le temps d'exécution global d'une application irrégulière.

Par conséquent, nous avons montré l'utilité d'intégrer un tel mécanisme dans le support d'exécution PM², qui représente le noyau commun sur lequel s'appuient les différents régulateurs de charge étudiés dans le cadre du projet ESPACE.

Dans les sections suivantes, nous détaillons le fonctionnement de ce mécanisme, ses limites, ainsi que son interface de programmation. Nous illustrons également sa facilité d'utilisation en donnant le code complet d'un petit régulateur élémentaire utilisant la migration pour équilibrer la charge.

5.4.2 Principe de la migration légère

Le support d'exécution PM² fournit quelques fonctionnalités permettant à une application[26] de provoquer la migration d'un processus léger (ou d'un groupe de processus légers) d'un *module* à un autre pendant son exécution. Cette opération, déclenchée par le simple appel à une primitive de la bibliothèque PM², provoque l'arrêt instantané de l'exécution du processus désigné, son transfert sur le module destinataire et la reprise de son exécution au point où elle avait été interrompue (figure 5.15).

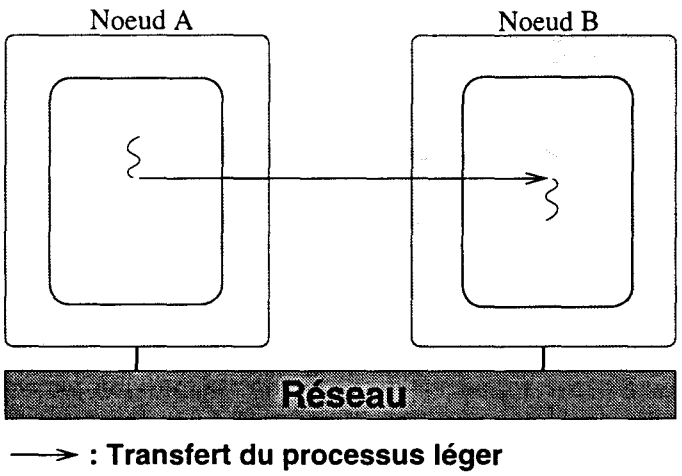

FIG. 5.15 - *La migration d'un processus léger provoque la continuation de son exécution sur un autre nœud.*

Le déroulement de l'opération est (normalement) transparent pour le processus léger déplacé et survient de façon asynchrone pour ce dernier. Le processus migré n'est donc pas tenu d'exécuter de quelconques traitements périodiques destinés à sauver son état ou à effectuer

26. Par exemple un régulateur de charge, qui est une application comme une autre du point de vue du support.

son « auto-migration ». Ce dernier point est important car, contrairement à ce qui se passe dans d'autres systèmes [115], la migration d'un processus peut réellement être initiée par un autre processus (par exemple un processus chargé d'équilibrer la charge entre les modules) et ne nécessite pas une *coopération active* de sa part.

De par sa nature, la migration d'un processus léger provoque uniquement le transfert de son contexte local (pile d'exécution, variables locales). Par conséquent, après sa migration, un processus léger se trouve plongé dans un nouvel environnement global (variables globales du module, variables « système » telles que les fichiers ouverts, etc.). Cette caractéristique est sans conséquence pour la plupart des applications développées avec PM^2, car le modèle de programmation ne préconise ni l'accès « direct » aux variables globales d'un module, ni l'accès (sauf sous contrôle) aux primitives de la bibliothèque C standard. Cependant, dans certains cas bien précis, il peut être nécessaire de *protéger* temporairement l'exécution d'un processus léger contre son éventuelle migration intempestive. C'est précisément ce point que nous allons examiner dans la section suivante.

5.4.2.1 Migrabilité des processus

Comme nous venons de l'évoquer, dans certaines circonstances, l'exécution d'un processus léger peut nécessiter l'accès à des données *situées en dehors de son contexte*. Typiquement, c'est le cas lorsqu'un processus sollicite des opérations d'*entrées/sorties* par le biais du système d'exploitation. Bien que l'on puisse considérer ces opérations comme « *marginales* » vis-à-vis du modèle de programmation PM^2, leur utilisation est parfois nécessaire dans certaines classes d'applications. Par exemple, les applications d'imagerie manipulent souvent des fichiers (images) de taille importante ne pouvant pas être chargés intégralement en mémoire vive. C'est pourquoi il est important de fournir des mécanismes assurant le bon déroulement de ces opérations dans un environnement où les processus sont susceptibles d'être migrés de manière asynchrone.

Le problème de l'accès aux données « globales » d'un module (y compris aux variables systèmes, donc) peut être illustré par la portion de code présentée en figure 5.16.

```
BEGIN_SERVICE(EXEMPLE)
   FILE *f;
   ...
   f = fopen("toto", "r");
   fread(buffer, size, f);
   fclose(f);
   ...
END_SERVICE(EXEMPLE)
```

FIG. 5.16 - *Exemple de code nécessitant une protection vis-à-vis de la migration.*

Dans cet exemple, une partie du traitement associé au service `EXEMPLE` concerne la lecture de données dans un fichier. Indépendamment du fait que les différents modules de l'application peuvent se trouver sur des nœuds connectés à des systèmes de fichiers différents, la suite d'instructions manipulant le descripteur de fichier `f` pose problème en cas de migration « intempestive » du processus léger l'exécutant, c'est-à-dire entre l'instruction `fopen` et l'instruction `fclose`. En effet, l'exécution de l'instruction `f = fopen("toto","r");` range dans la variable `f` une référence sur un ensemble d'informations (le *descripteur* du fichier « toto »)

stockées dans l'espace « noyau » du processus UNIX (*i.e.* le module) courant. L'utilisation de la variable `f` n'a donc de sens qu'au sein de ce module. C'est pourquoi la migration d'un processus léger exécutant cette portion du code s'avérerait catastrophique.

Il y a deux solutions possibles à ce type de problème :

1. Fournir une version « transparente à la localisation » de toutes les fonctions de la bibliothèque C standard.

2. Fournir un mécanisme simple permettant d'éviter à un processus d'être migré durant l'exécution de certaines portions de code.

La première solution est très lourde à mettre en œuvre, puisqu'elle nécessite la réécriture d'un grand nombre de primitives, et peu portable, car l'interface de la bibliothèque C standard présente quelques différences d'un UNIX à l'autre. De plus, à cause du principe même de transparence qui obligerait à exécuter la plupart des opérations à distance, elle conduirait à des performances d'exécution très pauvres. Enfin, cette solution ne peut s'appliquer qu'aux fonctionnalités de la bibliothèque standard et ne résout donc pas le problème dans un cadre général, en particulier lors de l'utilisation de bibliothèques de fonctions annexes.

C'est pourquoi la deuxième solution a été retenue dans le cadre de l'environnement PM², puisqu'elle présente une alternative à la fois générale, efficace et portable. Elle réside en l'existence de deux primitives, `pm2_disable_migration` et `pm2_enable_migration`, qu'il suffit d'appeler respectivement en début et en fin d'une portion de code « sensible » à la localisation. Leur utilisation est illustrée en figure 5.17.

```
BEGIN_SERVICE(EXEMPLE)
   FILE *f;
   ...
   pm2_disable_migration();
      f = fopen("toto", "r");
      fread(buffer, size, f);
      fclose(f);
   pm2_enable_migration();
   ...
END_SERVICE(EXEMPLE)
```

FIG. 5.17 - *Protection des portions de code sensibles à la localisation.*

Ces deux primitives fonctionnent réellement comme un « *parenthésage* » d'instructions et peuvent donc être utilisées de façon imbriquée : pour qu'un processus redevienne « migrable », il faut donc qu'il effectue autant d'appels à `pm2_enable_migration` qu'il a effectué d'appels à `pm2_disable_migration` (ce qui assure un fonctionnement correct lors de l'appel de fonctions en cascades).

Par défaut (et par mesure préventive) les processus légers commençant l'exécution d'un service ne sont pas migrables. Outre l'intérêt préventif, cela permet (en n'appelant pas `pm2_enable_migration` dans le service correspondant) d'éviter la migration des processus exécutant des traitements très courts, ce qui ne présenterait pas d'intérêt pour un régulateur de charge.

Pour en terminer avec la « migrabilité » des processus, signalons qu'une tentative de migrer un processus alors que celui est dans un état *non-migrable* provoque une exception

(CONSTRAINT_ERROR). Cependant, **cette situation ne peut jamais se produire** si l'on respecte les règles de bon usage des primitives de migration dans PM^2, comme il est expliqué dans la section suivante.

5.4.2.2 Migration et virtualisation

Nous l'avons vu, une application PM^2 est composée d'un ensemble de processus légers répartis dans plusieurs modules. Cet ensemble, typiquement de cardinalité importante (*i.e.* composé de centaines de processus légers), peut être caractérisé par un comportement extrêmement dynamique dans le temps. En effet, non seulement le nombre global de processus dans le système peut fluctuer à des fréquences élevées (applications irrégulières à parallélisme fin), mais l'état de ces processus peut également osciller entre *migrable* et *non-migrable*.

En n'oubliant pas que, dans PM^2, l'objectif de la migration est de fournir un outil permettant de rééquilibrer la charge à un instant donné, la question que nous nous posons maintenant est : « *Comment, dans un tel contexte, fournir une interface aux fonctionnalités de migration qui soit réellement utilisable ?* »

En fait, le problème principal consiste à trouver un moyen simple de permettre au programmeur de désigner, à un instant donné et dans un module donné (supposé surchargé), un ensemble de processus à migrer sur un autre module. Ce problème est difficile pour plusieurs raisons :

- Nous avons vu, dans une section précédente, que le mécanisme d'*appel de procédure à distance léger* est intéressant parce qu'il évite au programmeur de manipuler des identifiants de processus. Cette caractéristique, essentielle à une **virtualisation** réaliste de l'architecture, doit être conservée en présence d'un mécanisme de migration de processus. Il faut donc fournir au programmeur le moyen de désigner des processus « à migrer » tout en lui évitant de gérer lui-même la liste des processus dans le système.

- Si tous les processus légers d'une application « étaient égaux devant la migration », alors il suffirait de fournir une primitive permettant au programmeur de spécifier une simple *quantité* de processus à migrer et le système s'occuperait du reste. Hélas, la réalité est toute autre, car les processus présentent chacun des caractéristiques différentes (priorité, type de service exécuté, taille mémoire occupée, etc.). C'est pourquoi il faut permettre au programmeur d'**inspecter**, à un moment donné, les processus migrables d'un module et lui permettre de choisir, en fonction de leurs caractéristiques, les meilleurs candidats à la migration.

- Dans un environnement où les différents processus légers d'un module s'exécutent de façon concurrente (voire de façon réellement parallèle sur machines multiprocesseurs[27]), la désignation d'un processus léger est une opération *hasardeuse* par nature. En effet, pendant le temps s'écoulant entre la récupération du *nom* d'un processus et l'appel effectif à une primitive de manipulation (par exemple une opération de migration) de ce processus, le processus en question peut soit avoir changé d'état (il était migrable mais il ne l'est plus), soit même avoir disparu (son exécution s'est terminée).

27. Nous verrons dans le chapitre consacré à l'implantation de PM^2 que cette fonctionnalité n'est toutefois pas supportée dans la version actuelle de PM^2

En réponse à ces problèmes, nous proposons un mécanisme de migration de processus léger se décomposant en trois phases :

1. Dans un premier temps, l'appel à une primitive (`pm2_freeze`) *gèle* le module, c'est-à-dire stoppe complètement le déroulement des activités (processus) autres que le processus appelant la primitive, qui devient alors le seul processus actif dans le module.

2. Dans un second temps, l'appel à une primitive (`pm2_threads_list`) permet d'obtenir la liste de tous les processus légers *migrables* dans le module. Comme le système est gelé, le processus appelant peut alors calmement parcourir cette liste pour y sélectionner les processus (éventuellement aucun) qu'il décide de migrer.

3. Une fois ces processus choisis, dans un troisième et dernier temps, l'appel à une primitive (`pm2_migrate`) permet de déclencher le transfert des processus vers un module indiqué. Le module courant est alors dégelé et les autres processus reprennent leur activité normale.

Ces trois phases constituent, à notre sens, la meilleure manière d'utiliser des fonctionnalités de migration dans un environnement tel que PM². Dans une section à venir, nous montrons toute la puissance et la simplicité d'utilisation de ce mécanisme sur un exemple concret.

5.4.2.3 Informations de charge

Le mécanisme de migration de processus que nous venons d'étudier constitue un outil complémentaire à l'*appel de procédure à distance léger* en ce qui concerne l'équilibrage de charge des applications. En effet, alors que le LRPC est utile pour effectuer un placement des processus lors de leur **création**, la migration permet d'intervenir lorsqu'un déséquilibre est constaté lors de la **terminaison** ou du **blocage** de certains processus.

Pour être efficaces, ces deux mécanismes doivent être utilisés conjointement avec des fonctionnalités permettant d'évaluer la charge des modules à un instant donné. Dans ce but, le support d'exécution PM² fournit un certain nombre de primitives permettant d'obtenir, pour un module donné, le nombre de processus actifs, le nombre de processus bloqués (en attente de résultats de LRPC), etc. Nous ne détaillons pas ces primitives ici. Le lecteur intéressé pourra consulter les manuels d'utilisation des bibliothèques MARCEL et PM² [133, 134].

Typiquement, la réalisation d'un régulateur de charge pour une application conçue au-dessus de PM² s'effectue de manière totalement distribuée. Pour cela, une méthode couramment utilisée consiste en la création de processus légers spéciaux[28] (à raison d'un par module) chargés d'observer et, le cas échéant, de réguler la charge sur l'architecture. Ces processus sont créés au départ de l'application (par exemple à l'aide de LRPC asynchrones) et s'échangent périodiquement les informations de charge qu'ils collectent localement. À ce propos, nous verrons justement un exemple de régulateur fonctionnant suivant ce principe dans la section suivante.

Actuellement, cette « scrutation » de la charge des modules ne peut être effectuée que de manière périodique (*i.e.* en utilisant une instruction de temporisation). Bien que cette méthode permette d'obtenir de bons résultats ([52, 119]), elle présente le défaut d'avoir des performances liées à la période de scrutation choisie, elle-même dépendante des caractéristiques de l'application régulée.

28. que l'on appelle des *processus démons*

C'est pourquoi nous étudions actuellement des mécanismes permettant à un régulateur d'être automatiquement activé lorsqu'une variation de la charge significative est observée par le système. Ces mécanismes, qui mettent en œuvre des techniques d'« appels vers le haut » (*call-back functions*), permettront aux régulateurs:

- de réagir instantanément aux variations de charge locale;

- de ne pas perturber l'application de façon inutile.

Ces fonctionnalités sont cependant encore à l'étude et à l'heure actuelle nous n'avons pas encore vérifié leur intérêt pratique.

5.4.3 Exemple d'utilisation: petit régulateur de charge générique

Pour bien montrer la facilité d'utilisation du support pour la régulation de charge, nous allons maintenant étudier l'implantation d'un petit régulateur de charge générique[29] par migration de processus. L'objectif étant uniquement de montrer que la construction d'un tel régulateur est une opération assez aisée et non de discuter de la qualité de la régulation obtenue, nous avons choisi un algorithme de régulation assez simple opérant par lissage de la charge de proche en proche.

Le principe repose sur une organisation cyclique des modules, chaque module ne connaissant que ses deux voisins immédiats. Périodiquement, chaque module communique sa charge au module précédent. Chaque module peut ainsi comparer sa charge avec celle du module suivant et, le cas échéant, déclencher la migration de « la moitié de son excédent de charge » vers ce module. Ce principe de fonctionnement est résumé sur la figure 5.18.

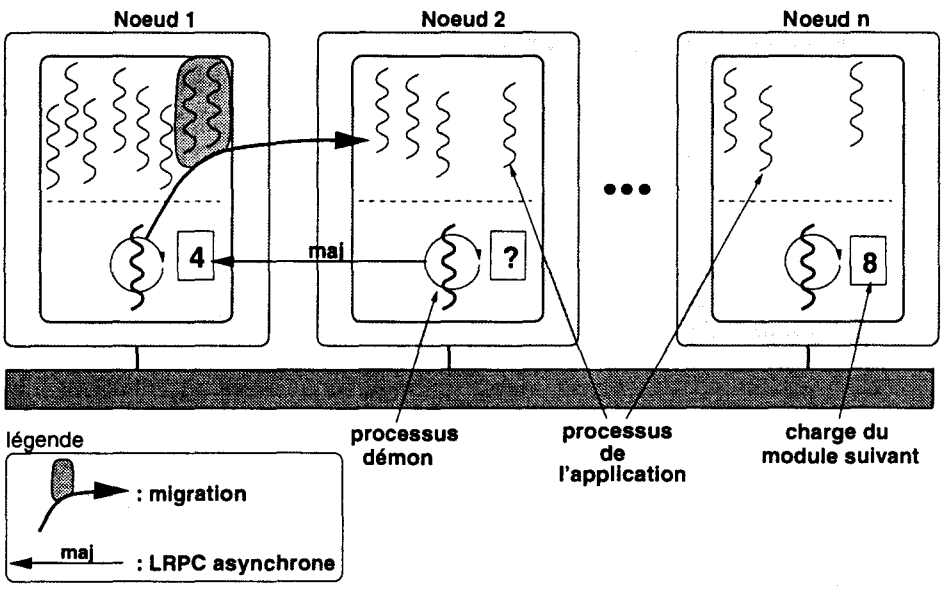

FIG. 5.18 - *Principe du régulateur « par lissage de proche en proche » de la charge.*

29. Au sens « indépendant de l'application régulée ».

L'implantation de ce régulateur de charge avec PM² repose sur l'utilisation de processus *démons* (un par module) effectuant périodiquement le travail décrit précédemment. Le code complet correspondant est donné en figure 5.19.

```
unsigned charge_du_suivant = 0;

BEGIN_SERVICE(INFORMATION_CHARGE)
   charge_du_suivant = req.charge;
END_SERVICE(INFORMATION_CHARGE)

unsigned charge_locale()
{
   return pthread_activethreads_np();
}

void reguler()
{ unsigned nb, quantite_a_migrer;
  pthread_t pids[MAX_THREADS];

   if(charge_locale() > charge_du_suivant) {
      quantite_a_migrer = (charge_locale() - charge_du_suivant) / 2;
      pm2_freeze();
      pm2_threads_list(MAX_THREADS, pids, &nb, MIGRATABLE_ONLY);
      pm2_migrate_group(pids, quantite_a_migrer, site_suivant);
   }
}

void informer()
{ LRPC_REQ(INFORMATION_CHARGE) info;

   info.charge = charge_locale();
   QUICK_ASYNC_LRPC(module_precedent, INFORMATION_CHARGE, &info);
}

void demon()
{
   while(1) {
      sleep(PERIODE_OBSERVATION);
      informer();
      reguler();
   }
}
```

FIG. 5.19 - *Exemple de régulateur utilisant la migration.*

En examinant ce code, on peut remarquer que la mise à jour des informations de charge s'effectue via le mécanisme d'*appel de procédure à distance léger asynchrone sans création de processus*, qui s'apparente dans ce contexte à un mécanisme d'accès mémoire distant (en écriture).

Outre sa compacité et son évolutivité, le gros intérêt de cette implantation consiste en sa relative indépendance de l'application régulée. En effet, même si le seul critère de régula-

tion pris en compte concerne le nombre de processus actifs par module, il est important de remarquer que le code de l'application peut ignorer l'existence de ce régulateur.

Bien entendu, certaines classes d'applications nécessitent d'être régulées selon des critères plus complexes que le nombre de processus actifs par module (par exemple l'occupation mémoire, les priorités des processus, etc.). C'est pourquoi des travaux sont menés au sein du projet ESPACE ([54]) dans le but de proposer une « bibliothèque » de composants logiciels, fournissant différentes stratégies de régulation et d'information, qu'il suffira de paramétrer au départ d'une application pour obtenir une régulation de charge personnalisée.

5.4.4 Migration et LRPC

Au début de ce chapitre, nous avons constaté l'intérêt du mécanisme d'*appel de procédure à distance léger* en tant que mécanisme principal d'expression du parallélisme dans un environnement dont l'ambition est de virtualiser l'architecture sous-jacente. De même, nous avons mesuré combien le mécanisme de migration était intéressant pour la construction de régulateurs de charge efficaces pouvant opérer de façon transparente[30] vis-à-vis des applications. Il nous reste maintenant à examiner la manière dont PM^2 gère la cohabitation entre ces deux mécanismes orthogonaux.

Comme nous l'avons déjà évoqué, l'exécution d'une application PM^2 se traduit par la présence de plusieurs arborescences dynamiques de processus légers étendues sur les différents nœuds d'une architecture distribuée. À un instant donné, on peut distinguer des processus « *intérieurs* » (*i.e.* des processus ayant généré des traitements dont ils attendront un retour de résultats) et des processus « *feuilles* » (*i.e.* n'attendant de résultat d'aucun autre). Il est facile de voir que, suivant la catégorie dans laquelle un processus se trouve, la migration « aveugle » de celui-ci n'aurait pas les mêmes conséquences sur le coût nécessaire au bon déroulement des mécanismes impliqués lors des LRPC. En fait, le problème se pose avec les processus *intérieurs* qui, s'ils sont migrés avant de recevoir les résultats qu'ils attendent, rendent plus coûteux le protocole d'acheminement des résultats (figure 5.20).

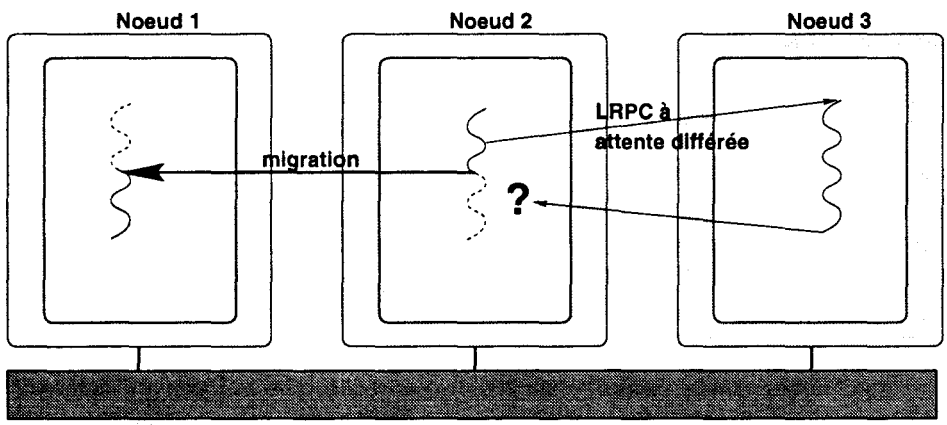

FIG. 5.20 - *La migration d'un processus entre l'appel d'une procédure distante et le retour du résultat pose problème.*

Ainsi, si les processus légers *feuilles* d'une arborescence peuvent être migrés sans que

30. Sauf en présence de pointeurs...

cela ne cause le moindre souci à l'environnement, il n'en est pas de même avec les processus *intérieurs*, qui se répartissent à nouveau en deux catégories :

Processus en cours d'appel synchrone Le processus est donc bloqué en attente des résultats du LRPC. Dans ce cas, il est marqué **non-migrable** par PM² jusqu'à l'arrivée des résultats. Nous pensons que ce choix est un choix « raisonnable » pour deux raisons. La première est qu'il est rarement utile de migrer un processus léger qui ne fait rien, comme c'est le cas ici, car l'utilité première de la migration concerne l'équilibrage de charge. La deuxième raison est que la migration d'un processus bloqué est un problème techniquement délicat à résoudre (les structures de synchronisation devant également subir des transformations post-migration), dont le coût opératoire ne se justifie pas dans ce cadre d'utilisation[31].

Processus ayant effectué un appel à attente différée Entre l'instruction d'appel de procédure et l'instruction d'attente des résultats, le processus est marqué **non-migrable par défaut.** La raison de ce comportement est conforme à la philosophie de PM², qui tente de faire en sorte que le coût de chaque primitive de son interface soit le plus transparent possible en ce qui concerne le nombre et la nature des opérations impliquées.

Dans ce deuxième cas, le comportement décrit est un comportement par défaut. Par conséquent, si le programmeur le souhaite, il peut le changer en indiquant, avant d'effectuer un tel appel, que le processus doit rester « dans le même état de migrabilité » entre l'appel et l'attente des résultats qu'avant l'appel. Cette indication s'effectue par un appel explicite à la primitive `pm2_keep_migratable(TRUE)`. Dans ce cas, le système permet une éventuelle migration du processus entre l'appel du service et l'attente des résultats, tout en assurant que, lors de l'opération d'attente correspondante, les résultats seront correctement récupérés. Dans le chapitre consacré à l'implantation de PM², nous verrons que ce mécanisme implique potentiellement des communications supplémentaires entre certains modules, mais que ce coût est constant et ne dépend pas du nombre de migrations que le processus *appelant* a éventuellement subies.

5.4.5 Contraintes techniques

La grande souplesse du mécanisme de migration proposé par PM² réside dans le fait qu'il permet de migrer des processus exécutant du code compilé par un compilateur C traditionnel[32]. En effet, contrairement à certains environnements imposant l'utilisation de langages [23] ou de compilateurs [98] spécialisés, PM² permet l'écriture des applications (en particulier le code associé aux services) en langage C classique[33], ce qui facilite la réutilisation d'applications existantes.

Malheureusement, toute médaille a son revers et le mécanisme de migration de PM² n'échappe pas à la règle. En effet, l'implantation de la migration ne pouvant pas bénéficier d'un quelconque pré-traitement effectué à la compilation, il lui faut s'accommoder de quelques contraintes incontournables. Pour bien comprendre les problèmes posés, nous allons tout

31. Par contre, la « migration sur disque » d'un processus bloqué est une opération très intéressante, qui peut justifier son coût opératoire. Nous y reviendrons en conclusion.

32. Nous verrons dans le chapitre consacré à l'implantation de PM² qu'il faut tout de même utiliser `gcc`. Ceci dit, `gcc` est un compilateur traditionnel...

33. Qui est le langage le plus utilisé sur système UNIX.

d'abord effectuer un survol des différentes opérations impliquées par une migration dans l'environnement PM^2.

5.4.5.1 Déroulement interne d'une migration

Du point de vue du système, un processus léger est principalement représenté par un *descripteur* et par une *pile d'exécution*. Le descripteur contient les informations nécessaires à la gestion du processus dans le système (priorité, taille de pile, sauvegarde des registres, etc.) alors que la pile d'exécution contient un empilement de *contextes* correspondant à la cascade d'appels de fonctions effectués jusqu'à la fonction actuellement exécutée par le processus léger.

La migration d'un processus léger consiste principalement à effectuer une recopie de ces deux entités (descripteur + pile) sur le site destinataire, puis à effectuer quelques traitements pour « adapter » le processus léger à son nouvel environnement.

Une description plus précise des étapes impliquées dans une migration est résumée sur la figure 5.21 :

- Dans un premier temps, le processus léger entre dans une phase d'« hibernation ». Ses caractéristiques sont alors recopiées dans un tampon de communication. Notons au passage que seule la partie « utile » de sa pile a besoin d'être recopiée, ce qui représente très souvent moins de 50 % de la taille de pile totale. Une fois cette recopie effectuée (qui constitue un condensé du processus), le processus initial est éliminé.

- Ensuite, le contenu du tampon est émis sur le réseau en direction du module destinataire de la migration.

- Lors de l'arrivée du message sur le module destinataire, un espace mémoire de taille suffisante pour stocker le descripteur et la pile du processus léger est alloué dans le système, puis ces données y sont recopiées. À cette étape, il est important de noter que la pile du processus peut ainsi être *relogée* à une adresse mémoire différente de la précédente (puisqu'il n'y a aucune raison que la cartographie mémoire des deux modules soit identique). C'est pourquoi certains traitements, sur lesquels nous reviendrons dans le chapitre consacré à l'implantation de PM^2, doivent éventuellement être effectués pour *adapter* les informations stockées dans la pile à ce nouvel emplacement. À la suite de cela, le processus léger est (enfin) réveillé, ce qui provoque son redémarrage au point précis où il avait été gelé.

Le principal avantage d'une telle réalisation du mécanisme de migration réside dans la simplicité des opérations impliquées et donc dans son efficacité (nous reviendrons sur ce point en section 6.4.4). Cependant, même si l'aspect technique des opérations décrites précédemment est maîtrisé, il n'en reste pas moins qu'un certain nombre de contraintes sont imposées par ce mode de fonctionnement.

5.4.5.2 Compatibilité binaire des modules

De par sa nature, la représentation en mémoire d'un processus léger (descripteur + pile) contient de nombreuses références dépendantes du code qu'il exécute. Par exemple, lorsqu'un processus effectue un appel de fonction, l'adresse de l'instruction suivant l'appel est sauvée dans la pile, de manière à pouvoir effectuer correctement le retour de la fonction par la suite.

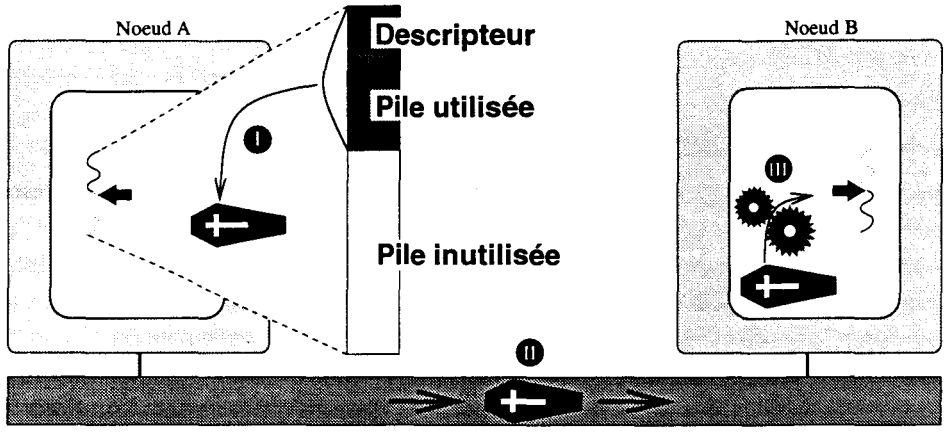

FIG. 5.21 - *Les trois étapes principales d'une migration.*

Cette *adresse de retour*, typiquement dépendante du fichier exécutable associé au module, n'a pas de sens dans un module qui ne soit pas strictement identique sur un plan binaire. La même remarque s'applique au *compteur ordinal* associé à chaque processus léger, qui figure également dans leur descripteur.

Il résulte de cette caractéristique qu'une migration de processus léger ne peut s'effectuer qu'entre **modules identiques** (même fichier exécutable, même architecture). *A priori* gênante dans un environnement prétendant exploiter des architectures hétérogènes, cette contrainte s'avère en concordance avec une réalité pratique :

1. Dans une configuration PM², les modules s'exécutant sur des nœuds de même nature n'ont pas de raison particulière d'être différents. En effet, le seul intérêt de constituer une application comportant des modules différents (implantations de services différents, ou implantations différentes de mêmes services) réside dans l'exploitation d'architectures hétérogènes pour optimiser l'implantation de certains services. Par conséquent, sur architecture homogène, l'immense majorité des applications PM² fonctionne suivant un schéma SPMD (Single Program Multiple Data).

2. Une configuration PM² hétérogène est souvent constituée de « groupes de nœuds » de même nature (*clusters* homogènes). Au sein d'un même cluster (machines multiprocesseurs telles que l'IBM SP2, la Ferme d'ALPHA DEC, etc.), les communications sont souvent beaucoup plus rapides (réseaux de fibres optiques, crossbars) qu'entre deux clusters distincts (réseau éthernet). De ce fait, il est plus « naturel » d'effectuer les opérations de migration, qui sont principalement destinées à un équilibrage fin de la charge, au sein des clusters.

Ainsi, l'organisation typique d'une configuration PM² sur une architecture hétérogène se décompose en un ensemble de *clusters*, chacun se décomposant à son tour en un ensemble de

nœuds homogènes connectés par un réseau rapide. Cette organisation à deux niveaux implique donc un équilibrage de charge à deux niveaux. Au sein de chaque cluster, les modules sont identiques, ce qui permet d'exploiter pleinement le mécanisme de migration pour équilibrer finement la charge entre les nœuds du cluster. L'équilibrage de charge entre les clusters s'effectue alors uniquement par placement de processus, ce qui est souvent raisonnable compte tenu des temps de communications induits par le réseau.

Cependant, cet équilibrage de charge à deux niveaux n'a pour l'instant été que partiellement évalué au sein de l'équipe, car l'exploration de l'équilibrage par « placement + migration » sur architectures homogènes constitue — pour l'instant — la priorité des travaux menés sur la régulation de charge au sein de l'équipe GOAL.

5.4.5.3 Références locales

Nous avons vu, lors de la description des trois étapes d'une migration, qu'un processus léger pouvait être relogé à une adresse mémoire différente de celle d'origine[34]. C'est pourquoi PM^2 doit souvent effectuer un traitement « post-migration » des piles et des descripteurs de processus légers. Cependant, certaines références situées dans la pile des processus peuvent échapper au contrôle de ce traitement. Il s'agit des variables de type « pointeur » déclarées au niveau applicatif.

Le problème posé par l'existence de telles références, non-détectables automatiquement sans l'aide d'un compilateur, réside dans le fait qu'elles peuvent contenir l'adresse d'un objet également situé au sein de la pile d'exécution (variable locale). Cette situation est illustrée sur la figure 5.22.

```
BEGIN_SERVICE(EXEMPLE)
   int entier;
   int *pointeur = &entier;
   ...
   (*pointeur) = 31;
   ...
END_SERVICE(EXEMPLE)
```

FIG. 5.22 - *L'utilisation des pointeurs pose problème vis-à-vis du mécanisme de migration.*

Dans cet exemple, la variable `pointeur` contient la référence d'un objet local situé dans le même contexte. Lors de l'exécution du code du service `EXEMPLE` par un processus, les instances des variables `entier` et `pointeur` se trouvent toutes deux dans sa pile d'exécution. Si le processus léger est migré durant l'exécution du service, par exemple juste avant l'instruction `(*pointeur) = 31`, alors il y a de très grandes chances[35] pour que l'adresse mémoire de la variable `entier` ne soit plus la même. Le contenu de la variable `pointeur` n'étant plus valide, l'instruction `(*pointeur) = 31` conduit donc à une exécution erronée.

Pour éviter ce type de comportement, l'environnement fournit un jeu de deux primitives (`pm2_register_pointer` et `pm2_unregister_pointer`) permettant au programmeur d'enregistrer dans le système les emplacements des pointeurs locaux aux fonctions exécutées par des processus légers. Dès lors, lorsque la migration d'un processus léger a lieu, le système (*i.e.*

34. *i.e.* celle où il se trouvait avant de subir l'opération de migration
35. Si l'on peut dire !

PM²) est capable de réajuster convenablement les références « utilisateur » situées dans la pile d'exécution du processus. L'utilisation de ces primitives est illustrée sur la figure 5.23.

```
BEGIN_SERVICE(EXEMPLE)
   int entier;
   int *pointeur = &entier;
   unsigned k;

   k = pm2_register_pointer(&pointeur);
   pm2_enable_migration();
   ...
   pm2_disable_migration();
   (*pointeur) = 31;
   pm2_enable_migration();
   ...
   pm2_unregister_pointer(k);
END_SERVICE(EXEMPLE)
```

FIG. 5.23 - *L'utilisation des pointeurs doit toujours s'effectuer dans un état* non-migrable.

Ce mécanisme ne suffit malheureusement pas à assurer une utilisation sécurisée des pointeurs « utilisateur » en présence de migration. En effet, non seulement les valeurs de tels pointeurs peuvent être stockées dans les registres du processeur, mais elles peuvent également être dupliquées dans la pile lorsque ces pointeurs sont passés en paramètre d'une fonction par exemple. Dans les deux cas, la localisation de ces références échappe totalement au contrôle du système, qui est donc incapable de les translater lors d'une opération de migration.

Par conséquent, il est nécessaire de protéger de manière systématique toutes les portions de code manipulant des adresses de variables locales, et ce pour prévenir d'éventuelles tentatives de migration. Cette démarche est illustrée sur la figure 5.23, où l'instruction `(*pointeur) = 31` est encadrée par les appels aux primitives `pm2_disable_migration` et `pm2_enable_migration`. Pour la même raison, il n'est pas conseillé, lors de la compilation d'applications PM² utilisant la migration, de spécifier des directives de compilation indiquant au compilateur d'effectuer des optimisations.

On le voit, le mécanisme de migration de processus légers fourni par PM² impose de lourdes restrictions sur l'utilisation des variables de type « pointeur » ; d'autre part les solutions apportées par l'environnement pour autoriser leur utilisation sécurisée ne sont pas complètement satisfaisantes, cela pour deux raisons :

- L'enregistrement des pointeurs de niveau « utilisateur » ainsi que la protection des portions de code manipulant des pointeurs est à la charge du programmeur, ce qui, pour certaines applications, peut alourdir considérablement la phase de conception ;
- Dans certains cas « pathologiques », c'est-à-dire dans une application constituée en majeure partie de code manipulant des pointeurs, les différents processus légers de l'application risquent d'être quasi-constamment dans un état *non-migrable*, ce qui peut nuire à l'efficacité d'un régulateur de charge par migration.

Pour toutes ces raisons, le modèle de programmation PM² préconise l'utilisation de pointeurs uniquement dans des situations d'extrême nécessité. Nous reviendrons sur ce point dans la conclusion de ce chapitre.

5.4.5.4 Données allouées dans le tas

En dépit du fait que le modèle de programmation PM^2 ne cautionne pas l'utilisation de pointeurs, à plus forte raison lorsqu'il s'agit pour un processus de désigner des variables situées en dehors de son contexte[36], certaines applications nécessitent l'utilisation de zones de données allouées dynamiquement dans l'espace mémoire global des modules. Dans ces applications, le code de certains services peut contenir des appels aux primitives de gestion de la *mémoire de tas* de la bibliothèque standard (*i.e.* `malloc`, `free`, etc.). Un exemple de tel service est illustré sur la figure 5.24.

```
BEGIN_SERVICE(EXEMPLE)
    char *pointeur;
    ...
    pointeur = malloc(strlen("bonjour")+1);
    ...
    strcpy(pointeur, "bonjour");
    ...
END_SERVICE(EXEMPLE)
```

FIG. 5.24 - *Exemple de service utilisant des données allouées dynamiquement.*

Bien évidemment, le problème de la migration se pose à nouveau pour ce type d'utilisation des pointeurs. Précisément, il s'agit de garantir qu'une migration de processus léger occasionne bien le déplacement des données que ce dernier a éventuellement allouées en mémoire globale[37].

Il y a plusieurs types de solutions à ce problème :

- Certains environnements, tels que UPVM [26], attachent à chaque processus un espace mémoire privé, lequel sera utilisé en tant qu'*espace de tas* pour toutes les opérations d'allocation dynamique de mémoire effectuées par le processus. Lorsqu'un processus est migré par le système, son espace mémoire privé l'est également. Selon les systèmes, cet espace mémoire peut se retrouver soit à une adresse de base différente, auquel cas une phase de translation de tous les pointeurs référençant une variable de ce tas est nécessaire [44], soit à une adresse de base inchangée, ce qui assure la validité des références existantes sur le tas [26]. Dans tous les cas, cette solution a l'avantage de permettre au système de maîtriser la gestion des opérations d'allocation. Cependant elle possède plusieurs inconvénients. D'abord, elle mène fatalement à une gestion des allocations gaspillant beaucoup de mémoire, car les *tas* sont pré-alloués. Ensuite, elle impose une limite arbitraire, potentiellement insuffisante, sur la taille mémoire disponible pour chaque processus léger. Enfin, elle augmente considérablement la mémoire occupée par un processus léger, ce qui, en outre, diminue l'efficacité des opérations de migration.

36. La désignation « brutale » de variables globales est normalement contraire à la philosophie PM^2, qui préconise un découpage des applications en processus légers n'interagissant que par l'intermédiaire de LRPC.

37. Précisons que ces données sont obligatoirement « à usage privé » et rappelons que, dans PM^2, le partage de données entre processus légers ne peut se faire que par l'intermédiaire de LRPC à vocation « d'accès mémoire à distance ».

- Certains environnements ne permettent pas la co-migration d'un processus léger et de la mémoire que celui-ci a allouée dynamiquement [115]. Dans ce cas, le processus migre sans une partie de ces données et la seule façon d'accéder à ces dernières et d'utiliser un mécanisme d'accès mémoire distant, dont la gestion reste à la charge du programmeur. Cette solution, qui est une solution de type « minimale », présente le principal inconvénient d'être très inefficace, puisque chaque accès mémoire *post-migration* devient un accès distant.

- Un environnement peut proposer un mécanisme empêchant la migration des processus utilisant des données allouées dynamiquement. Cette approche serait assez facile à mettre en œuvre dans l'environnement PM². Cependant, cela mènerait à des situations telles que nous les avons évoquées dans la section précédente, à savoir une proportion de processus non-migrables trop importante dans certaines applications, ce qui entamerait les performances des régulateurs par migration dans ce contexte.

La solution proposée par PM² pour résoudre ce problème favorise l'efficacité du mécanisme, tant sur le plan de l'occupation mémoire que sur le plan du coût d'exécution, en sacrifiant la transparence de sa mise en œuvre. Le principe réside dans la possibilité de définir, pour chaque type de processus léger (*i.e.* chaque type de service), deux fonctions qui seront appelées respectivement avant chaque migration (`pm2_set_pre_migration_func`) et après chaque migration (`pm2_set_post_migration_func`).

L'écriture de ces fonctions, qui reçoivent toutes deux l'identificateur du processus migré en paramètre, est à la charge du programmeur. Leur intérêt principal réside dans la possibilité de faire usage des différentes primitives d'empaquetage (resp. désempaquetage) fournies par PM². Dans ce cas, toutes les données ainsi empaquetées par l'utilisateur sont ajoutées au message de migration du processus (figure 5.21, phase 1) et, de façon symétrique, extraites du message lors l'arrivée du message sur le site (figure 5.21, phase 3).

Ce mécanisme, couplé avec les différents outils de l'interface POSIX Threads permettant de manipuler des variables « globales » à un processus léger (`pthread_setspecific` et `pthread_getspecific`), permet[38] donc de faire en sorte qu'un processus léger soit toujours migré avec les données qu'il a allouées dynamiquement.

5.4.6 La migration dans les autres environnements

La migration de processus n'est pas une idée nouvelle et plusieurs systèmes informatiques proposent des concepts s'y apparentant. Parmi ces systèmes, il convient de distinguer les systèmes d'exploitation des environnements de programmation parallèle.

La migration de processus implantée dans certains systèmes d'exploitation concerne uniquement les entités d'exécution de niveau « noyau », c'est-à-dire des processus lourds (Sprite [57], DEMOS/MP [142]) ou, dans certains cas, des processus de poids moyen (Mach [69]). Ces approches ne sont pas vraiment comparables avec l'approche PM², et ce pour deux raisons. La première est que, nous l'avons dit, cette migration ne s'apparente pas à de la « migration légère », ce qui implique des techniques de migration totalement différentes. Par exemple, la migration d'un processus lourd ne requiert pas la translation des différentes adresses qu'il contient, puisque celles-ci sont toutes exprimées de manière relative à son espace d'adressage. La deuxième raison est que la philosophie d'utilisation est totalement différente. En effet, dans

38. Mais c'est parfois pénible...

ces systèmes, les mécanismes de migration sont exclusivement déclenchés par le système d'exploitation lui-même, lorsque celui-ci en éprouve la nécessité (accès à des données distantes, équilibrage de charge).

Parmi les environnements de programmation parallèle autorisant la migration de processus, il est possible de dénombrer plusieurs domaines d'applications ainsi que plusieurs techniques de mise en œuvre. Nous présentons quatre environnements représentatifs des différentes tendances actuelles. Tous sont construits au-dessus du système UNIX :

Emerald [100] désigne un environnement distribué à objets ainsi qu'un langage parallèle à objets associé. L'objectif principal d'Emerald est de constituer une base pour l'expérimentation de la mobilité des traitements et des objets sur architectures distribuées (typiquement des réseaux de stations de travail). Dans cet environnement, les flots d'exécution sont pris en charge par des processus légers qui peuvent migrer d'une machine à une autre lorsqu'ils doivent exécuter une méthode sur un objet distant. Cette migration est totalement transparente pour le programmeur, sauf bien sûr du point de vue des performances.

Cette transparence totale, qui fait d'Emerald l'environnement le plus abouti dans ce domaine, est permise grâce à l'utilisation de techniques de compilation aidant le mécanisme de migration. D'ailleurs, des travaux récents dans ce projet permettent même la migration de processus exécutant du code natif en contexte hétérogène. C'est là toute la différence avec l'environnement PM^2, qui s'adresse à une classe d'utilisateurs beaucoup plus large en proposant la migration de programmes écrits en C ou même en Fortran, mais qui ne maîtrise pas le processus de compilation de ces applications.

MPVM (*Migratable PVM* [25]) est un environnement de programmation parallèle basé sur la même philosophie générale que PVM, avec lequel il est d'ailleurs « compatible ». L'objectif principal de MPVM est de fournir aux programmeurs d'applications parallèles utilisant PVM un outil permettant d'équilibrer automatiquement la charge de ces applications sur les différentes machines d'une architecture homogène. Pour ce faire, des processus *démons* espionnent régulièrement la charge des machines et déclenchent des migrations de processus PVM (donc lourds) lorsque cela s'avère nécessaire. À quelques restrictions près, l'opération de migration est transparente pour une application. Cette propriété a nécessité, de la part des concepteurs, une réécriture quasi-totale du mécanisme d'acheminement des messages PVM, de manière à *virtualiser* les identifiants de processus en les rendant indépendants de la localisation de ces derniers.

MPVM et PM^2 proposent un mécanisme de migration dans le même but, à savoir permettre un équilibrage de charge dynamique pour augmenter les performances des applications. Cependant, les similitudes s'arrêtent là. Dans PM^2, les mécanismes de migration de processus sont entièrement contrôlables par le programmeur et concernent des activités à grain fin (processus légers). Dans MPVM, la stratégie de migration des processus est intégrée à l'environnement et ne concerne que des activité à gros grain. Cette dernière caractéristique la rend très inefficace, puisque les auteurs rapportent typiquement des temps de migration de plusieurs secondes sur réseau éthernet.

UPVM [103, 26, 102], issus des travaux des concepteurs de l'environnement MPVM, constitue le « petit frère » efficace de ce dernier. La philosophie d'utilisation reste la même et conserve son interface à la « PVM ». De même, l'équilibrage de charge des applications

s'effectue par migration de processus et est intégré à l'environnement. La grande différence avec MPVM, c'est que l'entité d'exécution est le processus léger, encore appelée ULP (*User Level Process*). La migration de ces ULPs est complètement transparente pour les applications et s'avère logiquement plus efficace qu'une migration de processus UNIX.

Comme PM², UPVM a été implanté à partir d'une bibliothèque « maison » de processus légers de niveau utilisateur, de manière à pouvoir réaliser un mécanisme de migration de processus. Contrairement à PM², la migration dans UPVM ne nécessite pas de traitement post-migratoire de la pile d'exécution et ne pose pas de problème particulier avec les zones de mémoire allouées dynamiquement par les processus. Chaque fois qu'un processus (léger) est créé dans le système, un nouvel espace mémoire lui est réservé sur **tous les nœuds**, de façon identique (taille et adresse de base), et ce pour préparer d'éventuelles migrations futures. De cette façon, lorsqu'un processus léger est migré d'un nœud à un autre, il ne change pas d'adresse mémoire. D'autre part, chacun de ces processus contient son propre espace de tas privé, qui consomme évidemment une place importante. Ce tas, qui fait partie de l'espace mémoire du processus léger, est également pré-alloué sur tous les nœuds pour chaque création.

Malgré les avantages qu'elle comporte en terme de simplicité de mise en œuvre, cette approche possède deux inconvénients majeurs. Le premier est qu'une création de processus léger est une opération très coûteuse dans l'environnement UPVM, car elle implique une synchronisation globale entre tous les nœuds de l'architecture distribuée. Le second est que le nombre maximal de processus légers qu'il est possible de créer dans le système dépend uniquement de la taille mémoire disponible sur le nœud le plus chargé[39], ce qui, en plus d'entraîner un gaspillage énorme de la mémoire, constitue un obstacle au développement d'applications massivement parallèles.

Ariadne [115] est un environnement de programmation pour architectures multiprocesseurs (à mémoire commune ou distribuée). L'entité d'exécution de base est le processus léger et les applications privilégiées concernent le domaine de la simulation, où les processus légers se montrent particulièrement adaptés au grain des différents traitements concurrents. L'exploitation d'une machine à mémoire commune ou à mémoire distribuée n'est pas transparente et des fonctionnalités différentes peuvent être exploitées selon le cas. De manière commune aux deux types d'architecture, Ariadne permet une gestion assez « classique » des processus légers (création, destruction, changement de contexte). Lorsque la machine sous-jacente est une machine à mémoire commune, les processus légers doivent utiliser des mécanismes d'accès à une mémoire globale partagée pour communiquer ou pour se synchroniser. En contexte distribué, les possibilités d'interaction entre les différentes machines résident dans la création de processus légers à distance et dans la migration de processus. En particulier, cette dernière fonctionnalité constitue le seul mécanisme permettant à un processus léger d'accéder à des données distantes.

Malgré le manque d'« abstractions de haut niveau » dans le modèle de programmation d'Ariadne, cet environnement se distingue par la proposition d'un mécanisme de migration de processus légers (uniquement en contexte homogène) comme seul et unique mécanisme d'accès distant aux données. L'implantation de ce mécanisme est très similaire à celle adoptée dans la bibliothèque MARCEL, qui supporte la gestion des processus

39. *i.e.* disposant du moins de mémoire

dans l'environnement PM^2. Elle ne supporte cependant pas la migration des données allouées dynamiquement dans le tas. De plus, une différence importante existe au niveau de son utilisation, puisque dans Ariadne les processus légers doivent migrer de leur plein gré (auto-migration), alors que dans PM^2 ils peuvent l'être à l'initiative d'autres processus. Cette dernière caractéristique rend difficile la conception de régulateurs de charge pour l'environnement Ariadne.

5.5 Les appels de procédures à distance légers, et après?

Tout au long de ce chapitre, nous avons montré pourquoi le mécanisme d'*appel de procédure à distance léger* constitue un très bon outil de découpage des applications parallèles irrégulières. En particulier, nous avons montré qu'il constitue un opérateur de décomposition facilitant la parallélisation d'algorithmes séquentiels (approche *diviser pour régner*) et qu'il est adapté à une utilisation conjointe avec des mécanismes de migration de processus. À ce jour, de nombreuses applications ont déjà été développées à l'aide du support d'exécution PM^2 (cf. chapitre 7) et ont permis de vérifier les propriétés précédentes.

Cependant[40], le mécanisme d'*appel de procédure à distance léger* présente tout de même quelques faiblesses. Ces faiblesses ne constituent pas un défaut propre à ce mécanisme (on les retrouve, sous d'autres formes, dans les autres approches telles que celles basées sur l'envoi de message ou encore sur le partage virtuel de mémoire), mais sont plutôt intrinsèques à la manière d'appréhender le *découpage* de tâches en sous-tâches dans les environnements actuels. Nous reviendrons sur ce point par la suite, en examinant les deux domaines où ces « faiblesses » sont, à notre sens, les plus significatives.

Suite à ce constat, nous avons mené une réflexion axée sur l'apport de nouveaux mécanismes permettant d'apporter une réponse à ces problèmes, tout en restant conforme à la « philosophie PM^2 » pour les aspects relatifs à l'expression du parallélisme, à la virtualisation de l'architecture et à la mobilité des activités. Cette réflexion a débouché sur la proposition d'un nouveau mécanisme — le *clonage léger* — qui, comme nous le verrons par la suite, sans remettre en cause le mécanisme de LRPC[41], offre une alternative intéressante dans des cas de figure bien délimités.

Cette section est donc dédiée à la présentation de ce nouveau mécanisme dont l'intégration effective dans la plate-forme PM^2, précisons-le, est encore à un stade de prototype expérimental. De ce fait, le *clonage léger* ne bénéficie pas encore des retombées attendues dans le domaine applicatif.

La suite de cette section s'articule comme suit. Dans un premier temps, nous examinerons deux domaines dans lesquels le mécanisme d'*appel de procédure à distance léger* n'exhibe pas les qualités attendues, à savoir la *simplicité d'expression* du parallélisme inhérent à un algorithme dans certaines situations et l'*évolution des flots d'exécution* par rapport aux contraintes de précédence qu'ils doivent respecter. Ensuite, nous présenterons le mécanisme de *clonage léger*, en détaillant son interface d'utilisation et les avantages procurés par son utilisation. Enfin, nous examinerons comment les mécanismes d'*appel de procédure à distance* et de *clonage* peuvent être utilisés de façon complémentaire pour le développement d'applications parallèles et nous conclurons en précisant les limites de ce mécanisme et les perpectives à ce travail.

40. Et malgré tout le bien que nous pensons de ce mécanisme !
41. Ouf !

5.5.1 Faiblesses des LRPC

5.5.1.1 Expression du parallélisme

Malgré l'adéquation générale du mécanisme de LRPC à la décomposition parallèle d'algorithmes séquentiels, son utilisation introduit parfois certaines « lourdeurs » dans la phase de conception de l'application parallèle résultante qui se traduisent souvent par une sensible diminution de la lisibilité des fichiers sources.

Nous allons illustrer cette caractéristique sur un exemple simple, dont le pouvoir illustratif constitue d'ailleurs sa seule utilité[42] (figure 5.25).

```
Polynome P[100], Q[100]; /* constantes accessibles sur
                            tous les noeuds */
Coef f(Coef X)
{  Polynome D1, D2;
   int i;
   Coef S;

   S = CoefZero;
   for(i=0; i<100; i++) {
      derivee(P[i], D1);
      derivee(Q[i], D2);
      S = somme(S, quotient(eval(D1, X), eval(D2, X)));
   }
   return S;
}
```

FIG. 5.25 - *Exemple de fonction comportant une boucle pouvant être parallélisée.*

Dans cet exemple, il s'agit de paralléliser une fonction `f(X)` calculant, étant donné un certain nombre de polynômes P_i et Q_i ($0 \leq i \leq 99$), le nombre :

$$\sum_{i=0}^{99} \frac{\frac{\partial P_i}{\partial x}(X)}{\frac{\partial Q_i}{\partial x}(X)}$$

Pour simplifier le problème, nous considérerons que les polynômes P_i et Q_i sont accessibles sur tous les sites de l'architecture sous-jacente si celle-ci est distribuée.

Lorsque l'on observe la fonction `f`, il apparaît immédiatement qu'il est possible d'effectuer les appels `derivee(P[i], D1)` et `derivee(Q[i], D2)` en parallèle (deux par deux), en utilisant des LRPC à attente différée. C'est d'ailleurs la principale qualité de ce mécanisme que de rendre immédiate la parallélisation d'appels de procédures séquentiels indépendants. Dans le cas présent, il suffit donc (avec PM²) de définir un service `DERIVEE` avec une interface constituée d'un polynôme en entrée (ou plus simplement d'un indice de tableau et d'une valeur booléenne indiquant s'il s'agit du tableau `P[]` ou `Q[]`) et d'un polynôme en sortie, d'écrire les fonctions souches correspondant à l'empaquetage et au désempaquetage de ces polynômes, et de définir le corps du service consistant à un simple appel à la fonction `derivee` préexistante. Une fois ces opérations effectuées, on obtient une version parallèle de la fonction `f`.

42. Comprenez par là qu'il ne faut pas chercher de signification profonde dans la finalité de l'algorithme présenté.

Cependant, en examinant la fonction « de plus près », il apparaît que la parallélisation obtenue est faible, puisqu'elle n'exploite qu'un parallélisme de degré 2. Non seulement cette décomposition parallèle risque de ne pas être plus efficace que la version séquentielle (si le temps d'exécution de la fonction **derivee** est très court), mais surtout elle n'exploite pas du tout le parallélisme potentiel inhérent à la boucle **for**. En fait, il semble plus judicieux d'effectuer la parallélisation de **f** en cassant la boucle et en effectuant donc plusieurs « paquets » d'itérations en parallèle.

Sur la figure 5.26, nous montrons les grandes lignes d'une version parallèle de **f** basée sur un découpage de la boucle **for** en 10 morceaux[43] effectués en parallèle.

```
BEGIN_SERVICE(SOUS_CALCUL)
   int i;
   Polynome D1, D2;

   res.S = CoefZero;
   for(i = req.inf; i <= req.sup; i++) {
      derivee(P[i], D1);
      derivee(Q[i], D2);
      res.S = somme(res.S, quotient(eval(D1, req.X), eval(D2, req.X)));
   }
END_SERVICE(SOUS_CALCUL)

Coef f(Coef X)
{  int j;
   LRPC_REQ(SOUS_CALCUL) request;
   LRPC_RES(SOUS_CALCUL) result[10];
   pm2_rpc_wait wait_struct[10];

   request.X = X;
   for(j=0; j<10; j++) {
      request.inf = j*10;
      request.sup = j*10 + 9;
      LRP_CALL(module(), SOUS_CALCUL, STD_PRIO, DEFAULT_STACK,
               &request, &result[j], &wait_struct[j]);
   }
   S = CoefZero;
   for(j=0; j<10; j++) {
      LRP_WAIT(&wait_struct[j]);
      S = somme(S, result.S);
   }
   return S;
}
```

FIG. 5.26 - *Une parallélisation de la fonction* **f** *à l'aide des LRPC à attente différée.*

Dans cette version, un service SOUS_CALCUL a été défini. Celui-ci contient le code qui sera exécuté en parallèle par les différents processus légers de l'application, c'est-à-dire une boucle calculant la somme partielle associée à l'intervalle : $i \in$ [**req.inf..req.sup**]. Le corps

43. Le nombre 10 a bien sûr été fixé arbitrairement. En fait, le choix de tout autre diviseur de 100 aurait été possible. Cela n'aurait rien changé à notre propos.

de la fonction `f` consiste donc à effectuer 10 *appels de procédure à distance légers à attente différée* du service `SOUS_CALCUL`, en découpant l'intervalle [0..99] en tranches successives de 10 éléments ([0..9], [10..19], etc.). Enfin, lorsque tous les sous-résultats lui sont parvenus, leur somme est calculée, ce qui termine le traitement de la fonction `f`.

Sur un exemple aussi simple que celui que nous venons de voir, il apparaît une certaine « lourdeur » de mise en œuvre introduite par l'utilisation des LRPC. En effet, la définition du service `SOUS_CALCUL`, même si elle n'est pas complexe, est assez peu naturelle. Contrairement à la première version parallèle proposée où nous évoquions la définition d'un service `DERIVEE` (laissant augurer des possibilités de réutilisation ultérieure), cette deuxième version nous a obligé à inventer de toutes pièces une procédure `SOUS_CALCUL` spécialement adaptée à la parallélisation d'un code qui ne comportait pas un appel de procédure séquentiel correspondant à l'origine. Il a donc fallu identifier les paramètres de cette « nouvelle » procédure et écrire les fonctions « souches » associées, ce qui, compte tenu de la faible probabilité de réutilisation du service, s'avère assez frustrant.

Dans le cas présent, l'identification des paramètres du service `SOUS_CALCUL` était assez aisée, car la portion de code « parallélisée » utilise peu d'éléments appartenant au *contexte* de la fonction `f` (variable `X`). Cependant, il serait facile de trouver des exemples où une portion de code « parallélisable » utilise beaucoup de variables locales à une fonction. C'est par exemple le cas pour les portions de code situées au cœur d'un *nid de boucles*, où le calcul dépend souvent des valeurs courantes des différentes variables d'itération. En cas de parallélisation d'un tel code à l'aide de LRPC, il est nécessaire d'inclure toutes ces variables dans l'interface du service défini pour l'occasion, ce qui peut s'avérer laborieux si ces variables sont nombreuses et « dispersées » dans le contexte local de la fonction.

Suite à cette petite étude de cas, il apparaît donc que le mécanisme d'*appel de procédure à distance*, s'il est parfaitement adapté à la parallélisation d'algorithmes comportant, de façon native, des appels de procédure pouvant être effectués en parallèle, nécessite une mise en œuvre assez laborieuse dans le cas où il s'agit de paralléliser une portion de code quelconque, surtout si celle-ci fait usage d'une large quantité d'informations « contextuelles ».

Dans ce cadre, un mécanisme permettant par exemple à des processus légers, créés pour prendre en charge une partie du travail de leur « père », d'*hériter du contexte d'exécution de leur père* constituerait une commodité d'utilisation appréciable, voire un facteur d'amélioration des performances par rapport à une transmission « manuelle » de parties éparses de ce contexte[44].

5.5.1.2 Graphe de précédence et flots d'exécution

D'une manière générale, le mécanisme d'*appel de procédure à distance léger* constitue, du moins dans sa version « avec retour de résultat »[45], un moyen utilisable par un processus pour *déléguer* une partie de son travail à d'autres processus. Notons que ce phénomène de *délégation* n'est pas spécifique au mécanisme de LRPC et qu'il constitue le principe de fonctionnement de beaucoup d'applications parallèles utilisant des mécanismes tels que l'envoi de message ou encore la mémoire virtuellement partagée. Nous ne nous écarterons cependant pas du modèle de programmation PM² et ne discuterons donc que de l'*appel de procédure à distance léger*.

44. En contexte homogène, une recopie « en un seul tenant » de l'intégralité du contexte s'avérerait plus efficace qu'un ensemble de copies élémentaires d'un large sous-ensemble de ce contexte.

45. Dans la suite de cette section, nous considérerons que c'est toujours le cas.

Le problème que nous allons évoquer dans cette section concerne la gestion des ressources de l'architecture et est lié à l'utilisation d'un mécanisme de délégation pour le support des applications irrégulières. En fait, nous allons voir que la **dissymétrie** intrinsèque au modèle **client/serveur** est parfois responsable d'une mauvaise gestion des ressources du système[46] dans un contexte où la durée des tâches des applications n'est pas prévisible.

Pour cela, nous allons observer le comportement d'une application irrégulière (comportant un petit nombre de tâches) basée sur l'exploitation du mécanisme de LRPC. La figure 5.27 montre (à gauche) le graphe de précédence des différentes tâches de cette application.

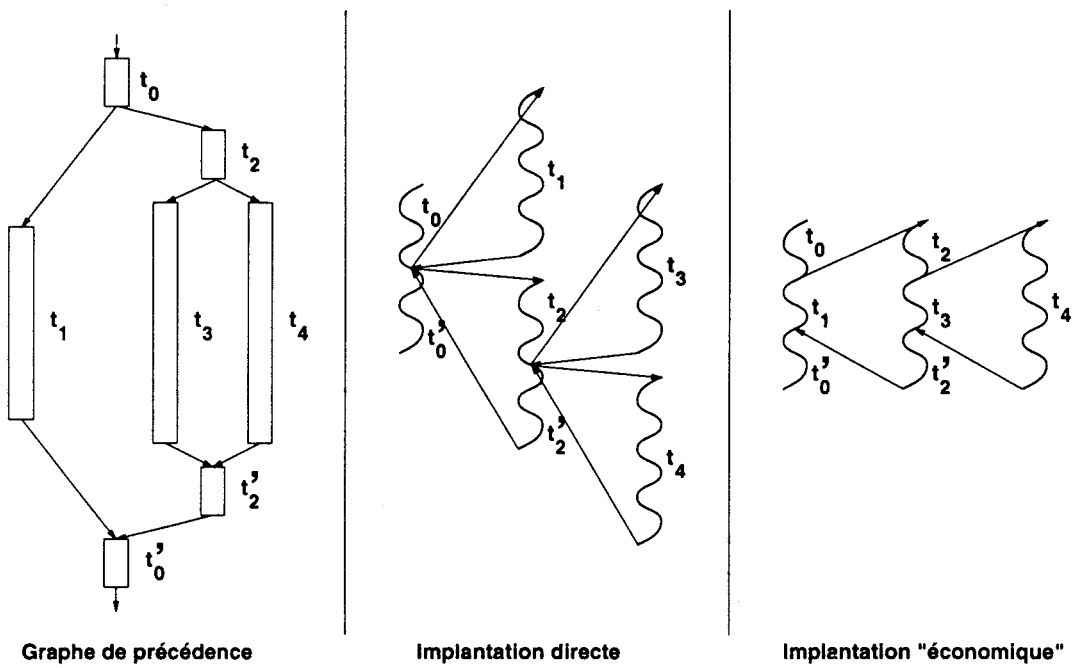

FIG. 5.27 - *À gauche : un exemple de graphe de précédence de tâches. Au centre et à droite : quelques implantations possibles avec des LRPC.*

Sur ce graphe, on observe par exemple que la tâche t_2 donne naissance à deux tâches exécutables en parallèle, t_3 et t_4, et qu'à l'issue de l'exécution de ces deux tâches, une tâche t'_2 peut commencer à s'exécuter. En réalité, dans un modèle basé sur l'*appel de procédure à distance*, les deux tâches t_2 et t'_2 n'en font qu'une. Plus précisément, elles représentent respectivement le traitement de début et le traitement de fin d'une même tâche. À un certain point de son exécution, la tâche t_2 crée deux sous-tâches t_3 et t_4, puis doit attendre la fin de leur exécution avant de pouvoir poursuivre la sienne, sous le nom de t'_2 dans le graphe de précédence. Le même raisonnement s'applique évidemment aux tâches t_0 et t'_0.

Notons que sur un plan théorique, rien n'empêche t_2 et t'_2 de constituer des tâches complètement différentes. C'est d'ailleurs le principe de fonctionnement des modèles d'*exécution dirigée par les données* (*DataFlow driven policy*). Nous y reviendrons ultérieurement. Cependant, d'un point de vue pratique, il s'avère qu'il est beaucoup plus facile (car plus naturel) de programmer de tels schémas de décomposition (*i.e.* sous-graphe compris entre t_2 et t'_2) à l'aide d'une même tâche s'exécutant de bout en bout, comme nous allons le voir maintenant.

46. En particulier en ce qui concerne l'occupation mémoire

L'implantation de cette application parallèle (*i.e.* dont le graphe est donné en figure 5.27) à l'aide du mécanisme de LRPC peut en fait s'effectuer de multiples manières. Cependant, ces implantations représentent toutes un « croisement » entre deux philosophies différentes :

Implantation directe L'implantation « directe » du graphe de précédence est illustrée sur la figure 5.27 (au centre). Lorsqu'une tâche peut se subdiviser en n sous-tâches, elle effectue n appels de procédure à distance à attente différée, puis elle se bloque en attente des résultats. Ainsi, dans l'exemple donné, l'exécution de l'application mène à la création de 5 processus au total. C'est typiquement l'approche que nous avons suivie dans l'exemple de la section précédente, lors de la parallélisation de la fonction `f`.

Cette approche présente l'avantage de permettre d'exploiter chaque « point de contrôle » du programme pour effectuer de la régulation de charge par placement de processus. Par exemple, lorsque la tâche t_2 génère les sous-tâches t_3 et t_4, l'exécution de celles-ci peut être effectuée sur d'autres nœuds de l'architecture. La seule contrainte dans ce domaine réside dans le fait que les tâches t_2 et t'_2 (resp. t_0 et t'_0), en l'absence de mécanismes de migration, seront exécutées sur le même nœud.

En revanche, une telle implantation présente l'inconvénient de mettre en œuvre un nombre important de processus (dans une application réelle) dont beaucoup sont inactifs (en attente de résultats) la plupart du temps. Pour revenir à notre exemple, on peut noter que tant que l'exécution des tâches t_3 et t_4 n'est pas terminée, les tâches t'_0 et t'_2 restent bloquées dans le système. En extrapolant ce résultat, il apparaît que dans les cas les plus défavorables, presque toutes les tâches d'une application peuvent être inactives à un instant donné. Compte tenu du fait qu'un processus, même léger, occupe une place mémoire non-négligeable (quelques dizaines de milliers d'octets) et que cette occupation est indépendante[47] de son état (actif ou inactif), une telle proportion de processus inactifs représente une utilisation maladroite[48] des ressources de la machine et peut porter à conséquence sur les performances générales de l'application (problèmes de *swap*).

Implantation économique C'est principalement pour atténuer l'inconvénient précédent que, la plupart du temps, l'implantation d'une application utilisant un découpage à l'aide de LRPC adopte un principe légèrement différent : lorsqu'une tâche peut se subdiviser en n sous-tâches, elle effectue $n - 1$ appels de procédure à distance à attente différée, puis exécute elle-même le code de la sous-tâche restante. Lorsque cette exécution est terminée, elle se bloque en attente des autres résultats. Cette approche est illustrée sur la figure 5.27 (à droite).

Par rapport à l'approche évoquée précédemment, il apparaît que, pour certaines applications, le gain en terme de nombre de processus utilisés peut être substantiel. Cela constitue un avantage non seulement en regard de la place mémoire occupée, mais aussi en regard du nombre d'opérations de création/destruction de processus dans les applications. En contrepartie, le placement des exécutions des tâches sur les nœuds de l'architecture est moins flexible (en l'absence de mécanismes de migration) et la programmation de l'application y perd souvent en lisibilité, car cette approche impose souvent de dupliquer le code des services dans le code des processus « appelants ».

47. Sauf, évidemment, s'il est terminé...
48. C'est un euphémisme !

Malgré le progrès effectué par rapport à l'approche précédente, le problème des processus inactifs n'a pas complètement disparu. Il est juste atténué. Cela est dû au fait que les applications traitées sont irrégulières, d'où l'impossibilité de garantir un découpage en sous-tâches nécessitant un temps de traitement équivalent. Ainsi, sur l'exemple de la figure 5.27, si la durée d'exécution de la tâche t_4 est grande par rapport aux durées d'exécution de t_1 et de t_3, alors les tâches t'_0 et t'_2 peuvent rester inactives dans le système pendant une durée potentiellement importante.

Comme on le voit, l'utilisation des LRPC pour le support des applications parallèles irrégulières possède l'inconvénient de mener à l'apparition de processus légers inactifs au cours de l'exécution d'une application. Au-delà des aspects relatifs à la consommation excessive de l'espace mémoire disponible sur l'architecture[49], ce phénomène, lorsqu'il est important, peut nuire à l'équilibre de la charge dans le système et perturber le fonctionnement de certains régulateurs de charge. En effet, des processus se bloquant (resp. se réveillant) introduisent autant (resp. plus) de *perturbations* dans le système que des processus se terminant (resp. venant au monde).

Pour éviter la présence de processus bloqués dans une application, certains environnements de programmation parallèle (Cilk [13], TPVM [66]) préconisent une exécution « dirigée par les données » des différentes tâches d'une application. Cependant, cette approche reste peu pratique à utiliser, car elle impose de concevoir une application comme un enchaînement de traitements non-bloquants, ce qui ne favorise pas une parallélisation aisée d'une application séquentielle préexistante.

En fait, en examinant pourquoi, même avec l'approche « *implantation économique* », certains processus deviennent inactifs, il apparaît que cela est dû à la conjonction de deux facteurs :

1. l'impossibilité de connaître à l'avance la durée des tâches générées ;
2. l'existence d'une dissymétrie entre les processus clients des LRPC et les processus serveurs.

Le premier facteur restant inéluctable, il reste la possibilité d'étudier un mécanisme permettant la suppression d'une telle dissymétrie entre les processus. Ce mécanisme permettrait alors de réellement diviser une tâche en n sous-tâches d'égale importance, tout en choisissant la tâche ayant la durée d'exécution la plus longue pour prendre en charge l'éventuelle suite de la tâche initiale. C'est précisément un tel mécanisme que nous décrivons dans la section suivante.

5.5.2 Le mécanisme de clonage léger

Dans cette section, nous introduisons la notion de *clonage léger* permettant d'exprimer le parallélisme inhérent à certains algorithmes de façon immédiate et efficace.

Nous présentons d'abord le principe de fonctionnement du mécanisme de *clonage léger*, puis nous présentons son interface et illustrons son utilisation sur un exemple, enfin nous exposons les principaux intérêts liés à son utilisation.

49. Ce problème peut être résolu, comme nous l'évoquerons en conclusion de ce chapitre, par des techniques de basculement en mémoire secondaire.

5.5.2.1 Principe

Le mécanisme de *clonage léger* est une opération permettant d'engendrer, de manière « temporaire », un certain nombre de processus *clones* à partir d'un processus léger initial. Deux processus *clones* l'un de l'autre possèdent, à l'instant suivant l'opération dite de *séparation*, un contexte d'exécution strictement identique. Entre autres, leur compteur ordinal référence la même adresse d'instruction machine et, surtout, leur pile d'exécution contient les mêmes données, héritées du processus ayant servi de *modèle* à l'opération de clonage.

En réalité, les différents processus issus d'une même opération de clonage possèdent chacun une donnée qui leur est propre : un numéro de rang[50] compris entre zéro et le nombre de clones moins un. En consultant la valeur de ce rang, il est alors possible à chaque clone d'exécuter un traitement différent des autres.

L'opération de *séparation* (qui déclenche le clonage) est comparable à l'opération *fork()* disponible sur les processus UNIX, à ceci près qu'elle s'applique à des processus légers et qu'elle est utilisée de façon beaucoup plus contrôlée. En effet, une deuxième opération, la *fusion*, doit être appelée par tous les clones d'une même génération pour mettre fin à une opération de clonage. Cette opération, effectuée de manière asynchrone par les différents clones, aura pour effet de ne laisser survivre que le clone ayant effectué cette opération le dernier (figure 5.28).

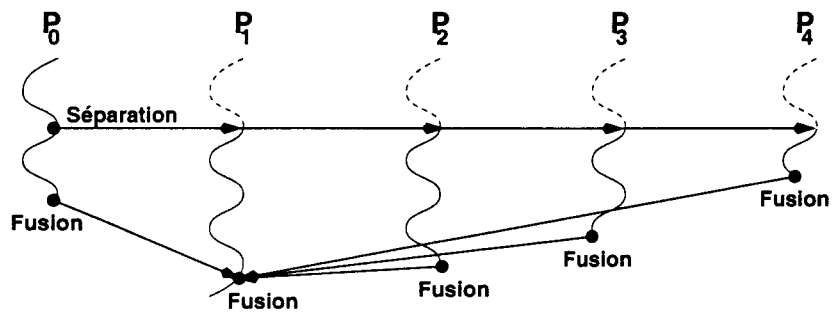

FIG. 5.28 - *Schéma conceptuel de l'opération de clonage. Lorsque qu'un processus exécute une opération de séparation, il est « cloné » en plusieurs exemplaires. Chaque exemplaire peut alors vivre sa propre vie, jusqu'à l'opération de fusion, exécutée par chaque clone de manière asynchrone. Le dernier clone exécutant cette opération sera le seul survivant.*

L'objectif du mécanisme de clonage étant de permettre le découpage d'un calcul en sous-calculs de manière symétrique, il est essentiel que les différents clones d'une même génération puissent apporter chacun leur contribution au calcul lors de l'opération de *fusion*. Voici donc le détail du déroulement typique d'une opération de clonage :

1. Lorsqu'un processus désire se subdiviser[51] en n exemplaires, il exécute une opération de **séparation** en indiquant le nombre de clones (en plus de lui-même) qu'il faut fabriquer, la liste des modules sur lesquels il faut les placer, le type du résultat qu'ils vont retourner lors de l'opération de fusion et l'adresse d'un tableau local pouvant stocker les résultats des différents clones.

50. Leur prénom en quelque sorte !
51. Ou plutôt devrais-je dire *se multiplier* ?

2. Le bloc d'instructions suivant une opération de clonage est donc exécuté par tous les clones. Cependant, en fonction de leur *rang*, ceux-ci peuvent manipuler des données différentes (par exemple en utilisant la pseudo-variable *rang* pour indicer des tableaux), ou même exécuter des instructions différentes.

3. Au bout d'un temps fini, chaque clone doit exécuter une opération de **fusion**, en fournissant la valeur du résultat de son calcul. L'appel à cette opération de fusion, pour des raisons de lisibilité, ne doit apparaître qu'une seule fois dans le code exécuté par les clones, de préférence dans le même bloc syntaxique que l'instruction de séparation correspondante.

4. L'instruction suivant immédiatement l'opération de fusion n'est exécutée que par un seul clone, qui constitue le seul survivant de l'opération de fusion. Ce processus peut alors accéder au tableau contenant les différents résultats communiqués par les clones.

5.5.2.2 Mise en œuvre dans PM^2 : interface de programmation

L'exploitation du mécanisme de *clonage léger* dans l'environnement PM^2 est très simple et réside dans l'utilisation de quatre primitives : `SPLIT`, `rank`, `MERGE` et `END_SPLIT` représentant respectivement les opérations de *séparation*, d'obtention du numéro de *rang()*, de *fusion* et de fin d'exploitation des résultats. Voici l'interface de ces quatre primitives :

`SPLIT(nb, tabmod, T)` C'est l'opération de *séparation*. `nb` représente le nombre de clones que l'on désire créer (il y aura donc `nb+1` clones au total). `tabmod` est un tableau d'identificateurs de modules spécifiant la localisation de chaque clone. Il doit donc contenir `nb` identificateurs. Enfin, `T` indique le *type* des résultats retournés par chacun des clones lors de l'opération de fusion.

`rank()` L'utilisation de cette primitive n'a de sens que si elle est appelée au sein d'une opération de clonage (entre une *séparation* et une *fusion*). Son appel permet à un clone d'obtenir son rang unique dans l'opération de clonage en cours (nombre de `0` à `nb`). Si le processus appartient à plusieurs opérations de clonage (clonages imbriqués), c'est le rang se rapportant à la dernière opération de séparation qui est retourné.

`MERGE(res, tabres)` C'est l'opération de *fusion*. `res` doit être l'adresse d'un élément de type `T` et représente la contribution du clone courant au résultat global de l'opération de clonage. `tabres` représente l'adresse d'un pointeur d'éléments de type `T`. Cette opération provoque la terminaison du clone appelant si celui-ci n'est pas le dernier à l'exécuter. Pour le dernier clone, elle retourne dans `tabres` l'adresse d'un tableau contenant les différents résultats récoltés lors de l'opération de fusion (ce tableau est donc de taille `nb+1`).

`END_SPLIT` Cette instruction signale la fin de l'exploitation des résultats de l'opération de clonage. Elle provoque la libération de la zone mémoire désignée par `tabres`.

5.5.2.3 Exemple

Nous présentons maintenant une illustration de leur usage en proposant une parallélisation de la fonction introduite en section 5.5.1.1 (figure 5.29).

```
Coef f(Coef X)
{  Polynome D1, D2;
   int i;
   Coef S;

   {  Coef R, *S_partiel;
      int from, to;

      SPLIT(9, modules(), Coef)
         R = CoefZero;
         from = rank()*10; to = from+10;
         for(i=from; i<to; i++) {
            derivee(P[i], D1);
            derivee(Q[i], D2);
            R = somme(R, quotient(eval(D1, X), eval(D2, X)));
         }
      MERGE(&R, &S_partiel)
         S = CoefZero;
         for(i=0; i<10; i++)
            S = somme(S, S_partiel[i]);
      END_SPLIT
   }
   return S;
}
```

FIG. 5.29 - *Parallélisation de la fonction* `f` *à l'aide du clonage léger.*

Comme nous le constatons sur la figure, une opération de clonage représente une portion de code parenthésée par les instructions `SPLIT` et `END_SPLIT`. Dans cet exemple, l'opération de séparation aboutit à la coexistence de 10 processus légers « clones ». Ensuite, chacun de ces clones calcule la somme partielle associée à son rang (intervalle $[10 \times rank() \,..\, 10 \times rank() + 9]$) dans une variable `R`. Enfin, chacun des clones exécute l'opération de *fusion* et le processus survivant effectue la synthèse de ces résultats (somme des sommes partielles).

5.5.3 Méthodologie d'utilisation conjointe avec les LRPC

L'utilisation du mécanisme de *clonage léger* pour exprimer le parallélisme inhérent à une application exhibe plusieurs avantages:

lisibilité du code On peut le constater sur l'exemple précédent (figure 5.29), l'utilisation de l'opération de clonage, comparativement à une utilisation des appels de procédure à distance légers, améliore considérablement la clarté du code source.

héritage de contexte Lors d'une opération de séparation, tous les « nouveaux » processus héritent de l'intégralité du contexte du processus « original ». Non seulement cette caractéristique évite au programmeur d'expliciter lui-même le transfert des informations de contexte utiles, mais en plus elle permet à chacun des clones d'une même génération de pouvoir « remplacer » le processus original lors de l'opération de fusion.

gestion des ressources Lors d'une opération de fusion, étant donné que tous les clones sont « égaux » devant cette opération, le système peut pleinement exploiter cette propriété en faisant en sorte que le clone effectuant le traitement le plus long constitue le processus « survivant » à l'opération. Moyennant un surcoût en terme de communications sur lequel nous reviendrons dans le chapitre 6, il est donc possible d'éviter le blocage à durée indéterminée[52] de processus dans le système, ce qui constitue un réel progrès en ce qui concerne la gestion des ressources mémoire de la machine.

En regard de tous ces avantages, et compte tenu du fait qu'une opération de clonage peut théoriquement être utilisée en lieu et place d'une opération d'*appel de procédure à distance léger*, il semble que la présence d'un mécanisme de clonage léger dans PM^2 rende le mécanisme de LRPC inutile. En réalité, il n'en est rien, et chacun possède en fait un domaine d'utilisation bien précis. En effet, compte tenu d'un certain nombre d'inconvénients et de limitations intrinsèques au mécanisme de *clonage léger*, son utilisation ne saurait être systématisée au cours du processus de décomposition parallèle d'une application :

1. D'un point de vue pratique, le mécanisme de *clonage léger*, puisqu'il consiste en une « *duplication* » de contexte d'un processus, impose à peu près les mêmes contraintes que le mécanisme de migration de processus. En particulier, il ne peut être utilisé que dans un **contexte homogène** et exige une compatibilité binaire stricte entre les différents modules impliqués dans une telle opération. De ce fait, comme pour la migration, il est limité à un usage sur un ensemble de nœuds de même nature, supportant l'exécution des mêmes modules (exécution SPMD). Pour cette raison, le mécanisme de LRPC garde tout son intérêt pour l'exploitation des architectures hétérogènes.

2. Dans certaines situations, les **informations de contexte** devant être transmises aux processus effectuant des sous-calculs sont **très peu nombreuses** et bien localisées. Typiquement, c'est le cas lorsque le code séquentiel consiste en une série d'appels procéduraux indépendants. Pour reprendre l'exemple du début (figure 5.25), cette situation est rencontrée si l'on décide de paralléliser les deux appels consécutifs à la fonction `derivee`, auquel cas la seule information de contexte à transmettre concerne le polynôme à dériver. Dans un tel cas de figure, il serait maladroit d'utiliser le mécanisme de *clonage léger*, car l'« héritage de contexte » entre les processus qu'il met en œuvre n'est pas justifié ici et risque donc de s'avérer moins efficace que l'utilisation des LRPC, par définition parfaitement adaptés à ce type de parallélisation.

3. Un deuxième argument vient justifier la maladresse d'utilisation du mécanisme de *clonage léger* lors de la parallélisation d'appels procéduraux consécutifs indépendants. Cet argument concerne l'**accroissement de l'espace de pile** utilisé par les processus légers. En effet, lorsqu'un processus léger effectue des appels procéduraux « en cascade » (la plupart du temps *récursifs*), son espace de pile disponible diminue en proportion[53]. Si la profondeur d'appel devient trop importante, la pile devient saturée. Dans un environnement tel que PM^2, un processus peut étendre sa pile dynamiquement, de façon à pouvoir faire face à ces appels en cascade. Cependant, comme nous le verrons dans

52. Par contre, des blocages dus à l'attente d'opérations d'accès mémoire à distance semblent difficilement évitables, comme il sera discuté au chapitre 6.

53. C'est bien entendu un abus de langage, car l'espace de pile occupé par un appel procédural varie en fonction de la procédure appelée.

la section 6.3.4, cette opération demeure assez coûteuse. C'est pourquoi il est souhaitable, lorsque c'est possible, d'éviter de longues chaînes d'appels récursifs dans le code exécuté par un processus léger. Or, dans le cadre de la parallélisation d'une portion de code constituée d'appels procéduraux consécutifs, les LRPC représentent justement une occasion de « briser » la récursivité en créant de nouveaux processus pour continuer le calcul. Au contraire, l'utilisation du mécanisme de *clonage léger* continuerait à augmenter la profondeur d'appel (pour chacun des clones) et rendrait de ce fait plus coûteuses les éventuelles opérations de clonage à venir.

À l'issue de cette section, il apparaît que le mécanisme de *clonage léger*, tout comme le mécanisme d'*appel de procédure à distance*, possède son domaine d'utilisation privilégié. Plus précisément, nous pensons que le développement futur[54] des applications avec PM² sera caractérisé par un découpage parallèle utilisant :

l'appel de procédure à distance lorsque la parallélisation concerne des appels procéduraux dans la version « séquentielle » du programme, ou lorsque le « contexte » à transmettre aux sous-tâches est petit ;

le clonage léger dans les autres cas.

Nous espérons ainsi que ce découpage mixte des applications offrira un bon compromis entre la lisibilité des fichiers sources, l'efficacité de la gestion des ressources et la capacité d'exploitation des architectures hétérogènes. Sur ce dernier point, il est clair que le mode d'exécution des applications sera organisé en « *clusters* » de modules identiques (même code, même architecture). Comme le mécanisme de migration, le mécanisme de *clonage léger* ne pourra s'appliquer qu'au sein de tels *clusters*, laissant aux LRPC le soin d'assurer l'équilibre des calculs entre les *clusters*.

5.5.4 Conclusion sur le clonage léger

Dans cette section, nous avons introduit un nouvel opérateur pour le découpage parallèle d'applications : le mécanisme de *clonage léger*. Nous avons vu que ce mécanisme a été conçu pour répondre à des besoins précis, qu'il semble assez prometteur dans cette voie et qu'il s'accommode somme toute assez bien de l'existence du mécanisme d'*appel de procédure à distance* dont il vient compléter les possibilités.

Il est important de noter, car nous l'avons à peine évoqué, que l'utilisation de ce mécanisme entre parfaitement dans la philosophie de conception prônée par le modèle de programmation PM², puisqu'il possède les mêmes qualités que le mécanisme de LRPC vis-à-vis des concepts de *virtualisation de l'architecture* (La gestion des processus est transparente), de *concurrence des calculs* (évidemment) et de *mobilité* de ces derniers (les processus clones sont parfaitement *migrables* si on les y autorise).

Dans le chapitre suivant, nous ferons le point sur l'implantation actuelle de ce mécanisme et verrons que les premières mesures de performance sont encourageantes. Cependant, comme nous l'avons annoncé en introduction de cette section, la « jeunesse » du mécanisme de *clonage léger* ne nous permet pas encore d'avoir le recul nécessaire permettant de juger son utilisation dans des applications en vraie grandeur.

54. Le mécanisme de *clonage léger* est très récent et n'a pas encore été expérimenté dans des applications « réelles ».

Il n'en reste pas moins que son étude demeure un sujet de recherche actif au sein de notre équipe, en particulier dans le cadre de nos collaborations avec d'autres équipes travaillant dans le domaine du *parallélisme de données*. Cette étude devrait nous permettre de dégager un certain nombre d'extensions à lui apporter de façon à pouvoir appréhender la manipulation de structures de données volumineuses (tableaux, matrices, etc.) attachées à chaque processus lors d'une opération de clonage, ou encore pour permettre le retour de données « non-contiguës » lors de la phase de fusion.

5.6 Conclusion

Dans ce chapitre, nous avons présenté le modèle de programmation proposé par l'environnement PM^2. Les principales fonctionnalités constituant la charpente de ce modèle sont clairement organisées autour de trois concepts :

Virtualisation PM^2 propose l'utilisation du mécanisme d'*appel de procédure à distance léger* pour la décomposition parallèle des applications. Le LRPC (*Lightweight Remote Procedure Call*) permet, sur la base d'une interaction de type *client/serveur*, la création d'un nouveau flot d'exécution pour la prise en charge d'un traitement spécifié. Ce flot d'exécution est directement supporté par un *processus léger* s'exécutant sur la machine cible.

Nous avons montré les multiples avantages d'une telle approche en ce qui concerne la facilité d'utilisation par un concepteur d'application et l'efficacité des mécanismes mis en œuvre à l'exécution. Mais surtout, nous avons mis en évidence l'intérêt de ce mécanisme dans une démarche de *virtualisation de l'architecture*, en montrant sa capacité à encourager les schémas d'exécution distribués et son très bon comportement dans un environnement supportant la mobilité des processus.

Concurrence PM^2 propose un modèle d'exécution basé sur un ensemble de processus légers distribués s'exécutant de manière concurrente. Pour cela, un *ordonnancement préemptif avec priorité* est effectué dans chacun des modules d'une configuration. Cet ordonnancement garantit une progression simultanée et équitable des différents processus d'un module, tout en respectant leur priorité d'exécution.

Nous avons montré qu'une telle politique d'ordonnancement est naturelle dans une démarche de virtualisation de l'architecture et qu'elle permet une plus grande souplesse d'implantation de certaines catégories d'applications (par exemple les régulateurs de charge). Mais surtout, nous avons montré que, contrairement aux idées reçues, cette politique peut contribuer à accroître considérablement l'efficacité des applications lorsqu'elle est associée à un mécanisme de régulation de charge par migration de processus.

Mobilité PM^2 fournit un ensemble de fonctionnalités permettant la conception d'applications dans lesquelles les différents processus peuvent être déplacés d'un module à un autre durant leur exécution. Contrairement à certaines formes de migration proposées par d'autres environnements, la migration des processus légers dans PM^2 est très souple (la plus souple à ce jour) et permet à un régulateur de charge de migrer des groupes de processus en quelques instructions seulement.

Au-delà de la souplesse du mécanisme, nous avons montré sa grande facilité d'utilisation en donnant le code complet d'un petit régulateur distribué utilisant la migration

de processus comme seul et unique moyen d'équilibrer la charge des modules d'une application. Nous avons également soulevé certains problèmes techniques rencontrés lors de l'utilisation de pointeurs dans le code utilisateur et apporté des solutions utilisables pour les contourner.

L'intérêt d'une association entre ces trois grandes familles de fonctionnalités a pu être vérifié dans plusieurs domaines applicatifs, comme il sera montré dans le chapitre 7. Depuis l'origine de l'environnement, les retombées de ces expérimentations ont eu une très grande influence sur l'évolution du modèle de programmation PM².

En cherchant à affiner davantage ce modèle, à la fois sur le plan de l'« utilisabilité » et sur celui de l'« efficacité », nous avons défini un nouveau concept — le *clonage léger* — permettant, dans des cas de figures précis, une meilleure expression du parallélisme ainsi qu'une meilleure utilisation des ressources à l'exécution. Une méthodologie d'utilisation conjointe avec le mécanisme de LRPC a été proposée, permettant l'intégration logique du mécanisme de *clonage léger* dans l'axe *virtualisation de l'architecture* du modèle PM².

Dans le chapitre suivant, nous montrons que l'implantation du modèle de programmation PM² peut être effectuée de manière portable sans pour autant porter atteinte à l'efficacité des différents mécanismes mis en œuvre.

5.6.1 Vers une virtualisation « totale »

Dans une démarche de *virtualisation totale* de l'architecture telle que l'ambitionne le projet ESPACE, l'environnement PM², qui en constitue la base, fournit un certain nombre de concepts et d'outils pour permettre la construction de *couches logicielles* s'approchant de ce but tout en se spécialisant dans certains domaines applicatifs. Nous l'avons dit, cette caractéristique impose (entre autres) à l'environnement PM² de fournir des mécanismes de gestion de processus capables de supporter un très grand nombre d'activités concurrentes.

Lorsque ce nombre devient très grand (*i.e.* plusieurs milliers de processus légers par nœud), le problème de la saturation de la mémoire physique des nœuds de l'architecture se pose. Dans la version actuelle de l'environnement PM², aucune limite autre que la quantité de mémoire *virtuelle* disponible n'est imposée quant au nombre maximal de processus légers sur un nœud. Aussi, lorsque toute la mémoire *primaire* d'un nœud est épuisée, le système d'exploitation (en l'occurrence UNIX) intervient en effectuant un va-et-vient (*Swapping*) entre certaines portions de la mémoire primaire et des zones en mémoire secondaire (zones de *Swap*). Ce mécanisme de va-et-vient est très souvent guidé par un mécanisme de pagination mémoire [4].

Comme nous le verrons au chapitre suivant, l'implantation de PM² est basée sur une bibliothèque de processus légers « de niveau utilisateur », ce qui rend l'existence de ces derniers complètement invisible pour le système d'exploitation. Cette caractéristique rend le mécanisme de va-et-vient du système totalement inadapté à la gestion mémoire d'un module PM², puisqu'il va fatalement *basculer* en mémoire secondaire des zones mémoire contenant les contextes d'exécution de certains processus légers, sans savoir s'ils sont actifs ou pas (figure 5.30). Étant donné que l'ordonnancement des processus dans PM² est un ordonnancement préemptif, cela risque même d'aboutir à un fonctionnement catastrophique consistant en un va-et-vient incessant entre la mémoire primaire et la mémoire secondaire à chaque changement de contexte.

C'est pourquoi nous envisageons d'intégrer à l'environnement PM² un mécanisme évolué de va-et-vient qui, lorsque la mémoire primaire sera saturée, sera à même de basculer

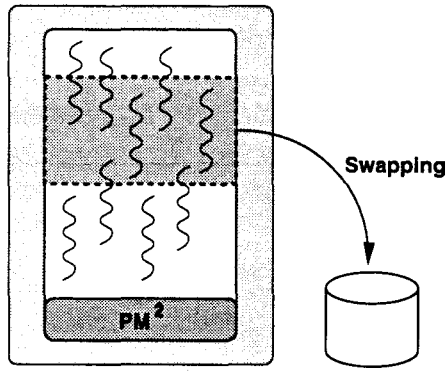

FIG. 5.30 - *Le mécanisme de « pagination » utilisé dans les systèmes d'exploitation s'avère totalement inadapté en présence de processus légers de niveau utilisateur.*

correctement certains processus en mémoire secondaire, éventuellement en retirant de la file des processus prêts ceux qui étaient actifs. La plupart des mécanismes de base nécessaires à ce « va-et-vient de processus légers » sont déjà disponibles dans la version actuelle de la plate-forme (cf. chapitre suivant). Néanmoins, le problème se situe davantage à un niveau conceptuel qu'au niveau technique, car il apparaît nécessaire d'accorder finement la stratégie de *va-et-vient* selon des critères tels que les dépendances inter-processus (basculement d'*arbres* de processus) ou encore la politique d'équilibrage d'un éventuel ordonnanceur global.

Il est difficile d'apporter une réponse précise à ces différents problèmes et leur étude constitue une perpective importante de ce travail.

Chapitre 6

PM^2 : réalisation et performances

Dans ce chapitre, nous décrivons les « grandes lignes » de l'implantation de PM^2, en ne détaillant que les aspects techniques les plus originaux de la plate-forme. La description de chacune des caractéristiques présentées est ponctuée par des mesures de performance.

Nous commençons par présenter l'architecture générale de la plate-forme PM^2 et expliquons les choix effectués en matière de bibliothèques de gestion de processus et de gestion des communications. Ensuite, nous décrivons brièvement la bibliothèque de communication utilisée, c'est-à-dire PVM (*Parallel Virtual Machine*). Puis nous détaillons la bibliothèque de processus légers « MARCEL » conçue et développée spécifiquement pour PM^2, qui constitue véritablement le cœur de la plate-forme. Cela nous mène enfin à l'étude du fonctionnement de PM^2, construit au-dessus de ces deux bibliothèques. Pour terminer, nous concluons sur la réalisation de l'environnement, en montrant l'intéressant compromis obtenu entre sa portabilité, son efficacité et les fonctionnalités qu'il offre.

6.1 Vue générale

La réalisation de l'environnement PM^2 s'appuie, comme c'est le cas pour la plupart des environnements basés sur une exploitation du parallélisme à grain fin en contexte distribué, sur une bibliothèque de communication et une bibliothèque de processus légers [132].

Lorsque nous avons démarré la réalisation de PM^2, compte tenu de l'objectif de portabilité que nous souhaitions atteindre, deux possibilités s'offraient à nous quant aux fonctionnalités de communication que nous allions utiliser.

La première possibilité consistait à développer PM^2 directement au-dessus de l'interface de programmation des « *sockets* UNIX ». Bien que cette démarche — adoptée par de nombreux environnements tels que Nexus [71], DTMS [35], PVC [42] — ait le mérite de permettre une bonne maîtrise de l'efficacité des mécanismes de préparation des messages [1], elle ne donne accès en revanche qu'à des fonctionnalités très rudimentaires et nécessite le développement de nombreux mécanismes supplémentaires pour assurer une utilisation « confortable ». Parmi ces mécanismes, on peut citer des fonctionnalités d'empaquetage de données, d'encodage/décodage des données en contexte hétérogène ou encore de « désengorgement » des *sockets* pour éviter certaines étreintes fatales [2]. Le développement de tous ces mécanismes représente un travail

1. puisque leur programmation reste à la charge du programmeur

2. Ce phénomène peut survenir lorsque deux processus s'envoient mutuellement de grandes quantités de données (assez grande pour saturer une *socket*), sans qu'aucun des deux ne prenne la peine de *lire* les données

considérable qui ne se justifiait pas dans le cadre du projet ESPACE, compte tenu du fait que les exigences de la plate-forme PM^2 en matière de communications étaient, à l'époque, relativement « standard ».

C'est pourquoi nous nous sommes orientés vers la deuxième possibilité, consistant à implanter PM^2 au-dessus d'une bibliothèque de communication « évoluée », telle que PVM [79], P4 [21], ou MPI [121]. À l'époque, l'interface MPI émergeait à peine et n'était pas encore normalisée. Au contraire, c'est la bibliothèque PVM qui faisait figure de « standard de fait » car elle était (et elle est encore) utilisée par une très large communauté de programmeurs d'applications scientifiques. De plus, nous avions déjà une première expérience de l'utilisation de cette bibliothèque dans un environnement à base de processus légers, dans le cadre du projet PVC [135]. En particulier, nous avions pu constater la grande souplesse d'utilisation de PVM (en particulier en ce qui concerne l'administration des configurations) et vérifier sa fiabilité[3]. C'est donc de manière assez naturelle que nous avons choisi cette bibliothèque pour la prise en charge des communications dans l'environnement PM^2.

Le choix de la bibliothèque de processus légers, en revanche, s'inscrit dans une démarche sensiblement différente de la précédente. En effet, contrairement aux environnements présentés au chapitre 4, les exigences de l'environnement PM^2 vis-à-vis des fonctionnalités des processus légers sont assez importantes (ordonnancement préemptif avec priorités, etc.). Certaines de ces fonctionnalités, comme la « migration »[4] de processus léger, ne sont pas réalisables au-dessus d'une bibliothèque n'offrant aucun support spécialisé. Or, à l'époque, aucune bibliothèque de processus léger existante ne proposait de tels services. C'est pourquoi nous avons conçu et développé notre propre bibliothèque de processus légers — nommée MARCEL — dans le cadre du projet ESPACE. Dans la section qui lui est consacrée, nous montrons qu'il est possible, en préservant un degré de portabilité élevé, de concevoir une bibliothèque de processus légers apportant des fonctionnalités évoluées (piles extensibles, processus migrables) tout en exhibant des performances de tout premier plan.

Ainsi donc, l'implantation de l'environnement PM^2 s'appuie sur la bibliothèque PVM pour la réalisation des communications entre les modules (LRPC, migrations, clonages) et sur la bibliothèque MARCEL pour la prise en charge de toutes les opérations relatives à la création, à l'ordonnancement et à la synchronisation des processus légers. Cette architecture est récapitulée sur la figure 6.1.

L'environnement PM^2 est actuellement opérationnel sur six architectures : Sparc/SunOS, Sparc/Solaris, Alpha/OSF-1, PC/Linux, PowerPC/Aix et Mips/Irix. Comme nous le verrons dans les sections à venir, le portage de PM^2 sur une nouvelle architecture ne nécessite qu'une (légère) adaptation de la bibliothèque MARCEL (en supposant la disponibilité de PVM bien sûr).

6.2 Gestion des communications : PVM

PVM (*Parallel Virtual Machine* [79, 111, 80]) est un environnement de programmation parallèle conçu au laboratoire national d'Oak Ridge (Tennessee) en 1989. L'objectif de cet environnement est de faciliter la conception de programmes parallèles exploitant des architec-

arrivantes pour débloquer la situation.

3. Étant donné la popularité de PVM, les « bogues » cachés ont du mal à subsister bien longtemps...

4. À ce niveau, nous devrions plutôt parler de « support pour la migration » puisque nous sommes en contexte local.

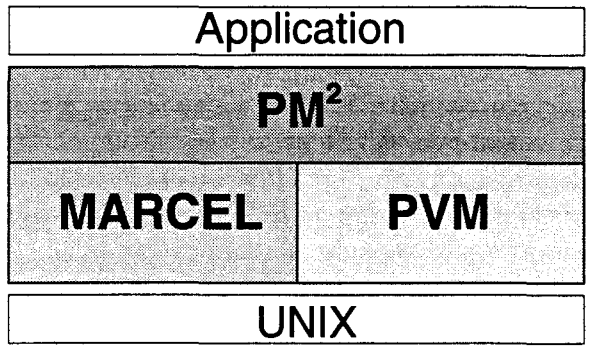

FIG. 6.1 - *L'implantation de PM² s'appuie sur une bibliothèque de communication (en l'occurrence PVM) et sur la bibliothèque* MARCEL.

tures distribuées éventuellement hétérogènes, et plus particulièrement des réseaux de stations de travail. Le modèle de programmation de PVM est basé sur l'envoi de message. Son modèle d'exécution repose sur un ensemble dynamique de processus lourds[5] communiquant par échanges asynchrones de messages. L'environnement PVM consiste en une bibliothèque de primitives (accessibles en langage C, C++ et Fortran) et en quelques programmes exécutables (démons, console).

Les deux principales qualités de PVM résident dans sa disponibilité sur un nombre impressionnant de systèmes[6] et sa capacité à gérer des configurations dynamiques avec une grande souplesse. Dans la suite de cette section, nous décrivons rapidement les « principales » fonctionnalités de PVM, relativement à l'utilisation que nous en avons faite. Cette description est uniquement destinée à permettre de mieux appréhender l'implantation de PM² décrite par la suite et n'est donc nullement exhaustive[7].

6.2.1 Machine virtuelle et nommage des processus

L'environnement PVM définit la notion de *Machine Virtuelle*, qui représente un ensemble de machines physiques reliées par la mécanique PVM. L'établissement d'une telle machine (*i.e.* la sélection des machines d'accueil pour une application) doit s'effectuer avant le lancement d'une application, mais pourra évoluer dynamiquement ensuite. À cet égard, l'environnement fournit un programme particulier, la *console PVM*, permettant de construire interactivement une machine virtuelle de manière incrémentale.

Sur chacune des machines d'une telle configuration s'exécute un programme en tâche de fond : le *démon* PVM. Ces démons sont virtuellement tous connectés les uns aux autres (en fait, ils communiquent par *sockets* via le protocole UDP) et sont à la base du fonctionnement du système. Entre autres, ils sont chargés du routage des messages entre les processus d'une application (dans le mode de fonctionnement par défaut) et des opérations de création de processus à distance.

Lorsque la machine virtuelle est configurée, des applications PVM peuvent être exécutées. Ces applications consistent en un ensemble de processus distribués sur la machine virtuelle.

5. UNIX la plupart du temps, mais il existe également des portages de PVM sur MS-Windows et OS/2.
6. On dénombre plus de 60 portages à l'heure actuelle, concernant plus de 30 architectures différentes.
7. Le lecteur intéressé pourra se reporter à [80] pour plus de renseignements.

Dans la terminologie PVM, ces processus sont appelés des *tâches*. Typiquement, le lancement d'une application s'effectue en exécutant d'abord « manuellement » un premier processus sur un des nœuds de la machine virtuelle. Ensuite, ce processus s'*enrôle* dans la configuration PVM en exécutant la primitive `pvm_mytid` qui lui renvoie un *identifiant* de tâche unique (entier sur 32 bits). Cet identifiant (`tid` – *Task IDentifier*) renferme la localisation du processus (identifiant du nœud hôte) et un numéro de série unique sur le nœud, et pourra être utilisé par d'autres processus lors d'opérations de communication.

Une fois le processus « enrôlé » dans PVM, il peut alors installer d'autres processus sur d'autres nœuds de la configuration, en utilisant simplement la primitive `pvm_spawn` en indiquant le nom du programme à exécuter et spécifiant éventuellement les nœuds d'accueil. L'exécution de cette primitive retourne les identifiants des nouvelles tâches ainsi créées, ce qui permet, tout au moins au processus créateur dans un premier temps, de leur envoyer des messages.

6.2.2 Communications

Le modèle de programmation PVM repose sur l'envoi de message asynchrone[8] entre les tâches d'une application, qui peut être de type « point-à-point » (`pvm_send`) ou de type « diffusion 1 vers n » (`pvm_mcast`). Malgré une utilisation sous-jacente du protocole UDP (par défaut), l'environnement assure que les messages (entre deux tâches) ne se doublent pas ni ne se perdent (grâce à un protocole de réémission en cas de perte par UDP).

6.2.2.1 Routage et transmission des messages

Comme nous l'avons évoqué précédemment, un des rôles principaux des démons PVM est d'assurer le routage des messages entre les tâches des applications. Notons que ce routage est simplifié par le fait que l'identifiant d'une tâche contient la référence du nœud (machine physique) sur lequel elle s'exécute. Lorsqu'une tâche émet un message, celui-ci est tout d'abord transmis au démon s'exécutant sur le même nœud (lien TCP). Ensuite, dans le cas où la tâche destinataire se trouve sur un autre nœud, celui-ci est transmis au démon se trouvant sur le nœud destinataire (lien UDP « fiabilisé »). Enfin, le démon délivre le message à la tâche destinatrice (figure 6.2).

Cette double[9] indirection peut s'avérer assez coûteuse à l'usage, principalement à cause des recopies de messages qu'elle occasionne. C'est pourquoi il est possible d'indiquer au système de privilégier certaines liaisons inter-tâches, en positionnant la stratégie de routage des tâches concernées sur `PvmRouteDirect` (ou tout au moins `PvmAllowDirect`). Dans ce cas, le système « profitera » des premiers échanges de messages entre ces tâches pour établir un lien TCP direct, qui court-circuitera l'itinéraire par démons interposés (figure 6.2). À la fin de cette section, nous illustrerons le gain de performance pouvant être escompté en utilisant cette stratégie.

De façon à simplifier l'explication que nous venons de voir, nous avons considéré que les messages étaient transférés en un seul tenant. En réalité, comme le protocole UDP impose une taille maximale quant aux trames circulant sur le réseau (4 096 octets), les messages sont automatiquement fragmentés en paquets par PVM lors d'une émission, puis réassemblés lors de la réception. La transparence de cette fragmentation, qui sollicite beaucoup les mécanismes

8. Nous verrons par la suite qu'en fait il peut être bloquant.

9. *simple* en cas de communication locale à un nœud

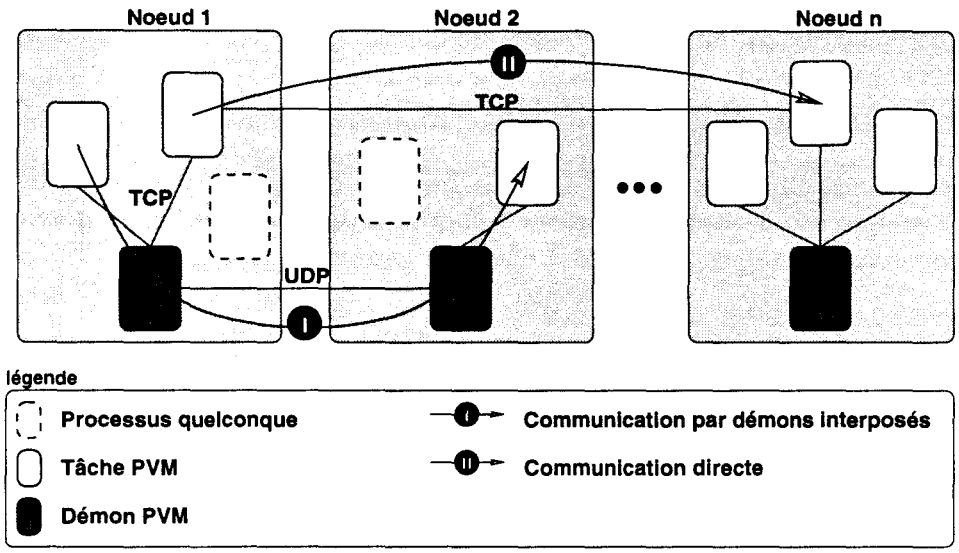

FIG. 6.2 - *Par défaut, PVM effectue un routage des messages par démons interposés. L'application peut néanmoins demander l'établissement de liens directs entre certaines tâches.*

d'allocation dynamique de mémoire, entre pour une grande part dans le relatif « surcoût » de la bibliothèque PVM par rapport aux mécanismes liés aux *sockets* UNIX.

6.2.2.2 Manipulation des messages

Afin de faciliter les opérations de transfert et de réception de données, la bibliothèque PVM permet de manipuler des objets de type « *tampon de communication* » dont la structure interne est cachée à l'application et qui représentent les seules entités diffusables entre les tâches. Les fonctionnalités associées à ces objets sont principalement[10] des opérations de création/destruction de tampons et d'empaquetage/désempaquetage des données dans les tampons.

La construction d'un message débute par la création d'un tampon (primitive `pvm_pmkbuf`) en spécifiant s'il est destiné à être envoyé à une tâche de même « format binaire » que la tâche émettrice (`PvmDataRaw` si oui, `PvmDataDefault` sinon). Si l'*encodage* spécifié est `PvmDataDefault`, alors toutes les données empaquetées dans le tampon seront converties puis stockées sous une représentation standard indépendante de toute architecture (en l'occurrence XDR de Sun [155]). De même, lors de leur désempaquetage, elles seront converties vers la représentation interne en vigueur sur la machine cible. Notons que l'encodage « XDR », lorsqu'il est spécifié, entraîne les conversions évoquées précédemment même si la tâche destinataire s'avère, au bout du compte, être de même format que la tâche émettrice. À la fin de cette section, nous examinerons le surcoût introduit par ces conversions par rapport à un encodage « brut ».

Une fois le tampon créé (et positionné comme tampon courant en émission), il est possible d'y insérer des données à l'aide de primitives d'empaquetages fournies pour chacun des types de base du langage C (`pvm_pkint`, `pvm_pkfloat`, `pvm_pkbyte`, etc.). Un message est construit

10. Il y a aussi des opérations de positionnement des tampons « courants » en émission et en réception, mais nous les évoquerons dans la sous-section suivante.

de façon incrémentale par appels successifs à ces différentes primitives, sans aucune possibilité de retour arrière ni de modification des données empaquetées. C'est là un des gros inconvénients de la gestion des tampons par PVM et nous y reviendrons dans la section consacrée à PM^2.

Des primitives duales de désempaquetage sont fournies pour extraire les données d'un tampon après réception (`pvm_upkint`, etc.). Les opérations d'insertion et d'extraction des données d'un message doivent être exécutées dans le même ordre.

6.2.2.3 Envoi et réception de messages

Lorsque la construction d'un tampon de communication est terminée, il est possible d'effectuer son émission vers d'autres tâches de l'application, via la primitive `pvm_send`. Cette primitive requiert deux arguments : l'identifiant de la tâche destinataire (`tid`) et le *type*[11] du message (nombre entier). Bien que l'opération d'émission de message soit *a priori* asynchrone (*i.e.* pas de synchronisation avec la tâche réceptrice), elle peut néanmoins bloquer la tâche appelante si le canal de communication sous-jacent (*socket* UNIX) est saturé.

Comme nous l'avons précisé précédemment, l'algorithme de routage des messages PVM assure que ceux-ci ne se doublent pas lorsqu'ils sont envoyés d'un même émetteur vers un même récepteur. En toute logique, la réception des messages en provenance d'une même tâche s'effectue donc dans l'ordre chronologique de leur émission. Ce comportement peut cependant être outrepassé en effectuant une *réception sélective*, en spécifiant un *type* précis de message attendu. D'autre part, il est également possible de spécifier un émetteur particulier lors d'une opération de réception, ce qui indique au système de mettre temporairement de côté les messages provenant de tout autre émetteur.

La primitive de réception de messages, `pvm_recv`, requiert donc également deux paramètres : l'identifiant de la tâche émettrice (-1 pour exprimer « *n'importe qui* ») et le type du message attendu (-1 pour exprimer « *n'importe lequel* »). L'appel à cette primitive est bloquant tant qu'un message correspondant aux spécifications n'a pas été reçu, ce qui peut s'avérer gênant dans certains cas. C'est pourquoi une version non-bloquante de cette primitive (`pvm_nrecv`, qui comporte les mêmes paramètres que `pvm_recv`), retournant un code d'erreur en cas d'échec, est fournie par PVM. Nous verrons dans les sections à venir l'importance de cette primitive pour autoriser la scrutation périodique des messages par un processus léger.

6.2.2.4 Mesures

De façon à mettre en évidence l'impact de paramètres tels que le choix du routage ou encore le choix de l'encodage sur les performances des communications, nous avons mesuré les temps de transmission de messages de différentes tailles entre deux tâches situées sur des machines reliées par un réseau rapide (débit maximum de 200 Méga-bits par seconde), et ce pour chacune des quatre combinaisons possibles. Les messages contenaient toujours des entiers et les temps relevés couvrent le déroulement complet d'une communication, c'est-à-dire sont mesurés depuis la création du tampon d'émission jusqu'à la destruction du tampon de réception. Les mesures ont été effectuées pour des tailles de messages variant de 256 à 16 384 octets et sont présentées en figure 6.3.

11. Nous verrons plus loin la sémantique de ce typage du message.

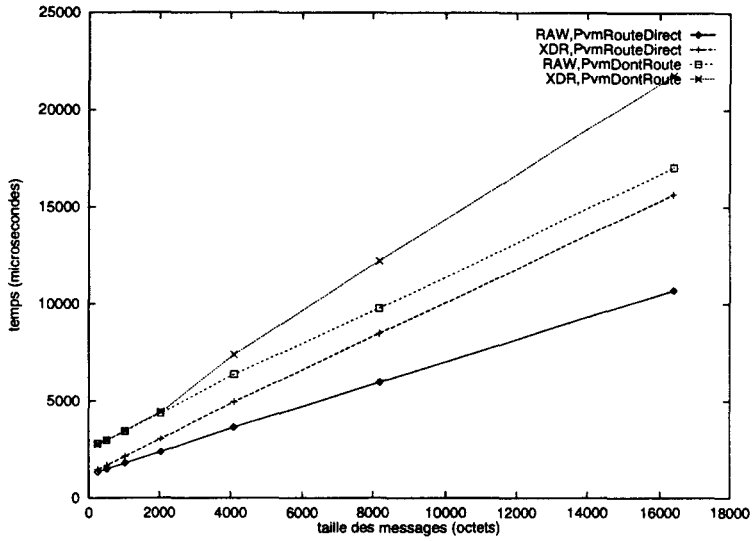

FIG. 6.3 - *Comparaison des latences de communication de PVM en fonction du routage et de l'encodage des messages entre deux processus distants (Processeurs ALPHA à 133Mhz reliés par Gigaswitch).*

Ces mesures mettent en évidence les deux phénomènes que nous évoquions précédemment.

1. Elles confirment le fait qu'un routage par démons interposés est beaucoup plus coûteux qu'un routage direct (jusqu'à 100 % plus coûteux pour les petits messages).

2. Elles montrent, de façon notable, que l'encodage XDR introduit un surcoût par rapport à un encodage brut. Cela montre qu'il est important de n'utiliser l'encodage XDR qu'en cas de nécessité, c'est-à-dire en contexte hétérogène.

6.2.3 Conclusion sur PVM

L'environnement PVM, de par sa souplesse d'utilisation et sa large disponibilité sur les architectures distribuées les plus diverses, constituait au début de ce travail une solution attrayante pour la réalisation des communications internes à un environnement tel que PM^2, d'autant que nous avions déjà une première expérience de ce type d'utilisation dans le cadre du projet PVC [135]. Un certain nombre d'environnements de programmation parallèle à grain fin ont d'ailleurs suivi une approche similaire [65, 20, 30].

Cependant, cette souplesse d'utilisation a un coût, qui se traduit principalement par un nombre important de recopies de données effectuées lors des transferts de messages, ainsi que par une gestion coûteuse de la mémoire (fréquentes allocations/libérations) occasionnée par les primitives de manipulation des tampons de communication.

Notons que sur certaines architectures, des versions « *propriétaires* » de PVM sont disponibles (IBM SP2, Cray T3D) et affichent des performances nettement supérieures à la version de PVM du domaine public. Ces performances restent cependant en retrait par rapport à des environnements tels que MPI et sont souvent obtenues par une restriction de l'interface originale de PVM et par conséquent une réduction de la souplesse d'utilisation.

6.3 Gestion des processus légers : MARCEL

MARCEL est une bibliothèque de gestion de processus légers en contexte utilisateur, dont la conception et le développement ont été effectués dans le cadre de cette thèse. Cette bibliothèque constitue la base principale de l'environnement PM^2 et celui-ci en est complètement dépendant.[12] L'objectif principal de MARCEL est de combler le vide laissé par les bibliothèques de processus légers existantes en matière de fonctionnalités avancées. La plupart de ces bibliothèques [127] [143] se conforment d'ailleurs à la norme « POSIX-threads »[99], qui représente le standard actuel en matière d'interface de manipulation des processus légers. Les raisons pour lesquelles l'utilisation d'une bibliothèque « strictement conforme à POSIX » ne convient pas dans le cadre du projet ESPACE sont nombreuses :

- La norme POSIX-threads propose avant toute chose la définition d'une interface aux processus légers pour la conception d'applications *portables*. La seule « contrainte » imposée aux applications réside dans le fait que cette interface n'est définie (pour l'instant) que pour le langage C. En revanche, la norme n'impose aucune restriction quant au compilateur employé pour générer le code exécuté par les processus légers d'une application.[13] Aussi les bibliothèques de processus légers « *POSIX-compliant* », conformément à cet esprit de portabilité maximale, ne fournissent aucune fonctionnalité dont l'implantation imposerait une technique de compilation particulière (ou tout au moins garantie). En particulier, c'est le cas pour des fonctionnalités telles que la migration ou le clonage de processus léger qui, comme nous le verrons par la suite, dépendent de la génération de code du compilateur.

  L'approche de la bibliothèque MARCEL consiste à céder un peu de terrain sur le plan de la portabilité, en imposant l'utilisation du compilateur C de la fondation GNU (c'est-à-dire `gcc`[14]), pour en gagner beaucoup plus (à notre sens) sur le plan des fonctionnalités.

- La norme POSIX-threads n'impose pas aux bibliothèques d'autoriser l'utilisation simultanée d'autant de processus légers que la mémoire disponible le permet. C'est pourquoi certaines de ces bibliothèques « de niveau noyau » fixent une borne maximale arbitraire à cette quantité (cf. implantation du noyau OSF-1). Dans le cadre du projet ESPACE, qui ambitionne une virtualisation totale de l'architecture, cette limitation est évidemment inacceptable.

  Dans les sections suivantes, nous verrons qu'en utilisant la bibliothèque MARCEL, une application peut comporter autant de processus légers concurrents que la mémoire de la machine le permet.

- Dans le même ordre d'idée, la norme POSIX-threads ne fournit aucune garantie aux applications quant à l'efficacité des mécanismes à la base de l'ordonnancement des processus. Là encore, dans un contexte où le nombre de processus légers par module peut être important, il est capital que l'ordonnanceur de ces processus garantisse des temps de changement de contexte indépendants du nombre de processus actifs.

12. PM^2 pourrait être porté sur d'autres bibliothèques de communication que PVM, mais pas sur une autre bibliothèque que MARCEL.

13. Ce qui peut d'ailleurs poser problème dans le cas de certaines optimisations de code effectuées par certains compilateurs. Mais cela sort du cadre de notre propos...

14. Qui constitue tout de même le compilateur le plus répandu, toutes architectures confondues.

La bibliothèque MARCEL contient un ordonnanceur de processus garantissant une durée *constante* des opérations de changement de contexte, quel que soit le nombre de processus légers présents dans la file des processus prêts et le nombre de niveaux de priorités autorisés.

- Comme toutes les normes, la norme POSIX-threads a des lacunes et nombreux sont les concepteurs d'applications réclamant de nouvelles extensions de l'interface [15]. Principalement, ces lacunes touchent au paramétrage fin des bibliothèques et concernent des fonctionnalités telles que la possibilité de modifier la durée des *quanta* de préemption, de paramétrer la taille d'éventuels « caches » de processus ou encore de fixer une taille arbitraire à une zone de mémoire « privée » à chaque processus. Certaines de ces fonctionnalités sont d'ailleurs fournies par quelques implantations ; elles sont reconnaissables aux noms des primitives associées, qui se terminent par « `_np` » (pour *not portable*). Dans des applications de calcul intensif, pour lesquelles tout doit être fait pour augmenter l'efficacité de l'exécution, de telles fonctionnalités peuvent jouer un rôle important.

  La bibliothèque MARCEL fournit actuellement de nombreuses possibilités de paramétrage. Grâce aux collaborations que nous menons avec les utilisateurs de PM^2, ces possibilités évoluent de version en version.

- L'interface proposée par la norme POSIX-threads est destinée à répondre à un large éventail de besoins tant au niveau applicatif qu'au niveau système dans un environnement de type UNIX. De ce fait, elle fournit (par exemple) des fonctionnalités de gestion des signaux analogues à celles disponibles pour les processus lourds. Ces mécanismes, qui n'ont pas d'utilité particulière dans les applications de calcul scientifique, ont cependant des répercussions sur les performances d'opérations telles que les créations ou encore les changements de contexte. [16]

  La bibliothèque MARCEL n'a pas une vocation aussi générale que les bibliothèques POSIX et son domaine d'utilisation, lié au projet ESPACE, concerne clairement les applications de calcul scientifique. Par conséquent, les primitives de gestion des signaux « à la UNIX » ne sont pas présentes dans cette bibliothèque [17], ce qui explique en partie (nous le verrons plus loin) les performances obtenues par rapport aux bibliothèques existantes.

Dans la suite de cette section, nous présentons les principales fonctionnalités faisant l'originalité de la bibliothèque MARCEL, en illustrant les techniques employées pour concilier *portabilité*, *efficacité* et *fonctionnalités*.

6.3.1 Extension de POSIX

De manière à ne pas imposer aux programmeurs une interface ésotérique supplémentaire dans le paysage actuel des processus légers, la bibliothèque MARCEL fournit une interface de programmation consistant en une *extension* de l'interface POSIX-threads. Autrement dit, les

15. cf. forum de discussion `comp.programming.threads`

16. Car il faut gérer des masques de signaux propres à chaque processus, qui nécessitent d'être positionnés lors d'une création et commutés lors d'un changement de contexte.

17. Notons qu'il est cependant possible de les définir dans une sur-couche de MARCEL, car le mécanisme de base nécessaire à une éventuelle gestion de signaux (la « déviation » de processus) est fourni.

fonctionnalités « génériques » relatives aux processus légers (création, terminaison, synchronisation, gestion d'attributs, etc.) adoptent une syntaxe conforme à POSIX. Toutes les autres (préparation à la migration, caches de piles, etc.) sont accessibles par des primitives signalées « *non-portables* » (`pthread_settimeslice_np`, `pthread_setdefaultstack_np`, etc.).

Encore une fois, il convient de nuancer l'expression « extension de POSIX-threads » puisque, nous l'avons évoqué dans la section précédente, certaines spécifications de l'interface POSIX-threads n'ont pas été implantées. La plus notable concerne la gestion des signaux UNIX. De même, plusieurs précautions doivent être prises lors de l'utilisation de la bibliothèque MARCEL, car elle ne fournit pas, du moins de manière transparente, une version sécurisée des primitives de la librairie C standard. En particulier :

- Les appels aux primitives d'allocation mémoire ainsi qu'aux primitives d'entrées/sorties standard doivent être effectués en exclusion mutuelle. Pour simplifier cette opération, une version « *thread-safe* » de quelques primitives d'usage courant sont fournies (`tmalloc`, `tfree`, `tprintf`, etc.).

- Les appels aux primitives d'entrées/sorties bloquantes risquent d'entraîner le blocage du processus UNIX tout entier. Si tel est le cas, le processus sera tout de même réveillé à la prochaine interruption d'horloge et un autre processus occupera le processeur, mais ce type de fonctionnement mènera à de piètres performances car à chaque fois que le contrôle sera rendu au processus léger en question, un quantum entier de temps processeur sera perdu. Pour éviter ces situations, la bibliothèque MARCEL fournit une primitive `tselect` (qui reprend la même sémantique que la primitive `select` d'UNIX) qu'un processus léger pourra appeler avant de tenter un appel bloquant.

Ceci étant dit, avant de passer à l'étude des mécanismes à la base de la gestion des processus, nous allons examiner une extension particulière proposée par la bibliothèque MARCEL en ce qui concerne la gestion d'erreurs : les exceptions.

6.3.1.1 Exceptions

Dans le langage C standard, la philosophie de rattrapage et de traitement des erreurs repose sur l'utilisation de conventions particulières encourageant l'utilisation de « codes de retour » pour signaler l'éventuelle occurrence d'une erreur au cours du déroulement d'une fonction. Lorsqu'un appel de fonction retourne un code d'erreur (généralement -1), il est alors possible de consulter la variable prédéfinie `errno` pour obtenir des renseignements plus précis sur la nature de l'erreur rencontrée.

Les inconvénients de cette approche sont nombreux et bien connus. En plus d'alourdir considérablement le code des applications (tests des cas d'erreur, propagation), elle ne place pas le programmeur à l'abri d'éventuels oublis de rattrapage d'erreurs. C'est pourquoi des langages plus modernes, tels qu'Ada [160] ou C++ [153], utilisent des mécanismes basés sur la notion d'*exception* pour le traitement des erreurs d'une application. Un des intérêts de cette dernière réside dans le fait qu'une exception non-rattrappée par l'utilisateur provoque l'arrêt de l'application, avec (dans le cas d'Ada) un message clair indiquant la localisation de l'instruction fautive.

Dans un contexte parallèle et distribué, la cause précise d'une erreur survenant à l'exécution est encore plus délicate à localiser que dans un contexte séquentiel [147]. Un mécanisme

tel que celui des exceptions peut aider à cette localisation. C'est pourquoi nous avons entrepris l'intégration d'un tel mécanisme dans l'environnement PM^2 (nous y reviendrons). Étant donné la dépendance de ce mécanisme vis-à-vis de la gestion des processus légers (les *levées* d'exceptions sont des opérations locales à chaque processus), il était nécessaire de le réaliser au sein de la bibliothèque MARCEL.

Principe Nous avons retenu une syntaxe et une sémantique similaires à celles utilisées dans le langage Ada83. La figure 6.4 illustre l'utilisation de ce mécanisme sur un exemple.

```
exception MY_ERROR = "MY_ERROR : exemple d'erreur définie par l'utilisateur";
...
f()
{
   BEGIN
      g();
   EXCEPTION
      WHEN(MY_ERROR)
         tfprintf(stderr, "Erreur bénine : le programme peut continuer\n");
      WHEN(STORAGE_ERROR)
         tfprintf(stderr, "Erreur fatale : plus de mémoire disponible\n");
         RRAISE; /* on propage l'exception */
   END
}
```

FIG. 6.4 - *Exemple d'utilisation du mécanisme des exceptions avec* MARCEL.

La *levée* d'une exception s'effectue au moyen de la primitive `RAISE(`*exception*`)`. Dans un traitant d'exception (*i.e.* après une instruction `WHEN(`*exception*`)`), il est possible de propager l'exception courante en invoquant la primitive `RRAISE` (cf. figure 6.4).

Comme dans Ada, la propagation d'une exception « remonte » les différents appels de fonctions jusqu'à trouver un traitant d'exception la concernant. Si la propagation d'une exception remonte jusqu'au niveau le plus haut (*i.e.* la fonction initiale du processus léger), alors le programme est interrompu par le déclenchement de l'erreur fatale `TASKING_ERROR` et un message indiquant le nom de l'exception, le numéro de ligne et le nom du fichier source où elle a été levée est affiché.

Implantation D'un point de vue technique, le mécanisme des exceptions est basé sur l'utilisation des fonctions C `setjmp` et `longjmp`. À chaque fois qu'un processus rencontre une instruction `BEGIN`, il établit un nouveau « point de saut » à l'aide d'un `setjmp` et chaîne ce point de saut avec le précédent. Lorsqu'une instruction `RAISE` est executée, un saut (`longjmp`) est exécuté vers le dernier point de saut rencontré et les traitants d'exception correspondants sont examinés. Si aucun de ces traitants ne concerne l'exception en cours, alors un nouveau saut est effectué vers le traitant précédent et ainsi de suite.

À chaque fois qu'un processus sort de la portée d'un bloc `BEGIN...END` (soit par le cours normal du programme, soit par une levée d'exception), le point de saut correspondant est « dépilé ». Étant donné l'absence de contrôle de la bibliothèque sur le compilateur, il n'est pas possible d'utiliser les instructions C `return`, `break` ou `continue` à l'intérieur de ces blocs,

car ils risqueraient de provoquer la sortie du bloc sans effectuer l'opération de dépilement du point de saut.

Dans la section concernant les opérations de translation des piles d'exécutions, nous verrons que ce chaînage des points de saut est nécessaire pour permettre au noyau MARCEL de parcourir la totalité de ces points lors d'un relogement du processus en mémoire.

6.3.2 Allocation des processus

En matière d'efficacité, l'expérience acquise lors du projet PVC nous a fait prendre conscience de l'influence considérable des mécanismes d'allocation mémoire sur les performances d'exécution des applications massivement parallèles. En particulier, nous avons pu vérifier combien il était important d'éviter une fragmentation mémoire excessive et, autant que possible, de réduire le recours aux primitives d'allocation au strict minimum.

C'est pourquoi un soin particulier a été apporté à la gestion mémoire effectuée par la bibliothèque MARCEL, compte tenu de l'utilisation intensive des processus légers dans l'environnement PM^2. En particulier, la création d'un nouveau processus léger[18] ne requiert l'allocation que d'une seule zone de mémoire qui contiendra :

- la pile d'exécution du processus ;
- le descripteur du processus ;
- un espace mémoire privé au processus.

L'adresse de base de cette zone mémoire constitue l'*identifiant* d'un processus (type `pthread_t` dans la terminologie POSIX). Toutes les informations relatives à un processus léger sont stockées dans cette zone. Comme il n'existe aucune structure globale de taille fixe répertoriant la liste des processus du système, le nombre de processus légers pouvant exister simultanément est uniquement limité par la quantité de mémoire disponible. Cette organisation est illustrée en figure 6.5.

Le *descripteur* d'un processus léger contient tous les attributs du processus (priorité, taille de pile, point de saut courant, etc.), une zone de données permettant de sauver son contexte d'exécution à chaque commutation et un certain nombre de variables systèmes à usage interne. Par exemple, le descripteur contient des pointeurs qui permettront son insertion dans une file de processus (par exemple la file des processus prêts de l'ordonnanceur) sans nécessiter l'allocation d'une structure annexe.

La pile d'exécution du processus contient son contexte d'exécution, c'est-à-dire un empilement de blocs correspondants aux appels de fonctions en cours. Lors de la création d'un processus léger, juste avant l'exécution de sa première instruction, le registre de pile du processeur est positionné de manière à pointer en début (*i.e.* en haut) de la pile du processus. C'est l'**unique opération** de toute la bibliothèque qui nécessite d'être écrite en assembleur (ce qui représente 2 à 3 mnémoniques tout au plus). Nous reviendrons sur l'organisation de la pile d'un processus dans la section consacrée à l'extension des piles.

Nous allons maintenant examiner la « raison d'être » d'un espace privé pour chaque processus.

18. En supposant qu'il ne peut pas être alloué dans le cache

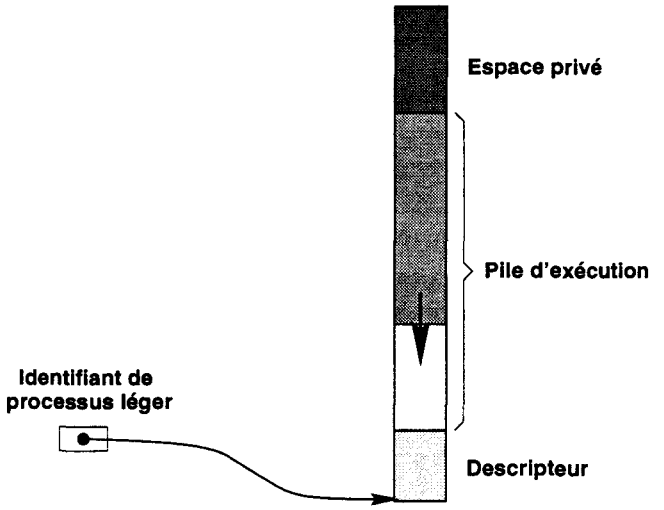

FIG. 6.5 - *Les données d'un processus léger sont toutes contiguës et peuvent donc être allouées en une seule opération.*

6.3.2.1 Espace mémoire privé alloué à la création

Dans l'interface POSIX-threads (et donc également dans l'interface MARCEL), la primitive de création d'un processus léger (`pthread_create`) accepte trois paramètres en entrée qui sont respectivement un pointeur sur une structure décrivant les attributs du processus (priorité, taille de pile, etc.), l'adresse de la fonction à exécuter et un argument (de type `void *`) qui sera passé en paramètre de la fonction. S'il s'avère nécessaire de communiquer plusieurs arguments lors du lancement d'un processus, il faut donc recourir à l'une des techniques suivantes :

1. Si les arguments à transmettre au processus sont déjà alloués de façon durable en mémoire et qu'ils sont regroupés dans une même structure, alors il suffit de passer l'adresse de cette structure en argument au processus.

2. Si, par contre, les arguments ne sont détenus que temporairement par le processus appelant, alors la première possibilité consiste à recopier ces arguments dans une zone allouée dans le tas et de passer l'adresse de cette zone au processus. Le processus devra alors se charger de la libération de la zone lorsqu'il n'aura plus besoin de ces données.

3. Pour éviter l'opération d'allocation dynamique du point précédent il est possible, dans le cas où la taille de ces arguments est connue à la compilation, de procéder ainsi :

   - Dans un premier temps, il faut regrouper ces données dans une structure contenant par exemple un sémaphore de synchronisation, créer le processus en lui passant l'adresse de cette structure, puis se bloquer sur le sémaphore.
   - Le processus nouvellement créé doit commencer par recopier les données de la structure vers son espace de travail local (puisque la taille était connue à la compilation). Lorsque c'est fait, il peut signaler la fin de l'opération en réveillant le processus appelant via le sémaphore.

- Une fois réveillé, le processus appelant peut réutiliser la structure comme bon lui semble.

Dans un contexte d'utilisation tel que celui de l'environnement PM^2, les processus légers sont créés pour prendre en charge l'exécution de fonctions associées à des *services*. Lors de leur création, il est donc nécessaire de leur transmettre un certain nombre de paramètres, comprenant non seulement les arguments du LRPC, mais également des données « systèmes » telles que l'identifiant du nœud où il faudra retourner les résultats, etc. Cependant, aucune des trois solutions évoquées précédemment ne convient pour réaliser cette transmission : la première parce que tous ces arguments ne sont évidemment pas stockés de manière durable, la seconde parce qu'une allocation dynamique est coûteuse en temps d'exécution et poserait problème en cas de migration (cf. section 5.4.5.4) et la troisième parce que la taille de ces arguments n'est pas connue à la compilation.

Pour ces raisons, nous proposons une extension de l'interface POSIX-threads permettant de spécifier la taille, lors de la création d'un processus, d'un espace de données adjacent à sa pile d'exécution (et donc alloué par la même occasion). Voici la suite d'opérations à effectuer pour réaliser le transfert d'un nombre quelconque de paramètres à un processus léger lors de son démarrage :

1. Tout d'abord, il faut positionner dans la structure décrivant les attributs du processus, la taille de la zone désirée (`pthread_attr_setuserspace_np`).

2. Ensuite, l'appel à la primitive `pthread_create` s'effectue comme d'habitude, à ceci près que, étant donné qu'une taille non-nulle a été spécifiée pour la zone spéciale, l'exécution du nouveau processus ne démarre pas à cet instant (il est uniquement alloué en mémoire).

3. Il reste au processus appelant à récupérer l'adresse mémoire de cette zone puis à y recopier les arguments du processus (`pthread_getuserspace_np`).

4. Enfin, lorsque les arguments sont prêts, on peut autoriser le démarrage du nouveau processus (`pthread_run_np`).

L'usage de cette fonctionnalité, intensivement utilisée par l'environnement PM^2, est récapitulé sur la figure 6.6.

6.3.2.2 Caches de processus

Toujours dans le but d'améliorer les performances des applications en diminuant le nombre d'opérations d'allocation dynamique de mémoire, il est intéressant de mettre en place un mécanisme de « recyclage » des processus légers. Dans les environnements bâtis au-dessus de bibliothèques n'offrant pas une telle fonctionnalité, il est possible d'effectuer ce recyclage en constituant un *pool* de processus « usagés » (bloqués sur un mécanisme de synchronisation quelconque) de manière à pouvoir, le cas échéant, réveiller l'un de ces processus au lieu d'en créer un nouveau. Bien que cette approche présente l'avantage d'être très efficace (la « création » d'un nouveau processus peut se réduire à une simple copie de paramètres et à une insertion dans la file des processus prêts), elle possède l'inconvénient d'être spécifique à l'opération de création d'un nouveau processus et ne saurait être utilisée dans le cas d'une opération de migration ou de clonage.

```
pthread_t pid;
pthread_attr_t attr;
char *ptr;
...
   pthread_attr_init(&attr);
   pthread_attr_setuserspace_np(&attr, strlen("bonjour")+1);
   pthread_create(&pid, &attr, fonction, NULL);
   pthread_getuserspace_np(pid, &ptr);
   strcpy(ptr, "bonjour");
   pthread_run_np(pid, ptr);
```

FIG. 6.6 - *Illustration du mécanisme de transmission d'arguments lors de la création d'un processus léger.*

Pour cette raison, compte tenu du fait que le coût d'une opération de création dépend principalement du coût de l'opération d'allocation mémoire sous-jacente, la bibliothèque MARCEL utilise un mécanisme de *cache mémoire* pour optimiser les opérations d'allocation de processus. Ce cache, de taille paramétrable par le programmeur (`pthread_setstackcache_np`), contient les « piles »[19] de processus ayant terminé leur exécution.

De façon à garantir un temps d'allocation minimal dans le cache, toutes les piles stockées dans celui-ci ont la même taille, qui est, à tout moment, égale à la taille de pile « par défaut » en vigueur. Par conséquent, lorsqu'un processus léger termine son exécution, sa pile n'est stockée dans le cache que si elle est égale à la taille de pile par défaut (et que si le cache n'est pas plein). Lorsqu'une allocation de pile doit avoir lieu (création, migration, clonage) et que sa taille correspond à la taille de pile par défaut, le cache est inspecté et une pile en est retirée (s'il n'est pas vide). Notons que la taille de pile par défaut peut être changée dynamiquement (`pthread_setdefaultstack_np`), ce qui a pour conséquence de vider complètement le cache de son contenu.

Afin de mettre en évidence le gain obtenu par une allocation dans le cache par rapport à une allocation classique (`malloc`), nous avons mesuré les temps d'exécution d'une opération de création de processus (avec une taille de pile par défaut) dans chacun des deux cas de figure. L'application test consiste en une boucle effectuant deux fois la création de plusieurs processus légers (20) suivie de leur terminaison. Lors de la première itération, tous les processus sont alloués dans la mémoire de tas. Lors de la seconde itération, les processus de la version « avec cache » du programme sont tous alloués dans le cache (de taille 20). Les résultats (tableau 6.1) montrent les temps de création d'un processus lors de la deuxième itération.

Clairement, le gain d'une telle optimisation varie fortement suivant le système d'exploitation et l'architecture.

6.3.3 Ordonnancement des processus

En plus du mécanisme d'allocation des processus, les performances d'une application à base de processus légers dépendent principalement de l'efficacité de son ordonnancement. Cette efficacité se rapporte principalement aux opérations relatives aux commutations de

19. C'est un abus de langage. En fait, cela désigne toute la zone mémoire contenant les données d'un processus (descripteur + pile + espace privé).

TAB. 6.1 - *Efficacité comparée des temps d'exécutions d'une opération de création de processus léger (en microsecondes) avec ou sans préallocation dans le cache.*

	Temps de création	
Architecture	Allocation dans le tas	Allocation dans le cache
PC/Linux 120Mhz	129 μs	13 μs
Sparc 5/Solaris 85Mhz	46 μs	35 μs
Sparc ELC/SunOs 25Mhz	378 μs	327 μs

contexte entre processus et, dans une moindre mesure, aux opérations d'insertion et de retrait dans les différentes files de processus maintenues par l'ordonnanceur.

Les dernières opérations mentionnées — insertion et retrait — sont sous-jacentes aux opérations de création, de destruction et de blocage des processus légers. Dans le contexte d'utilisation de PM², les créations sont liées aux appels de LRPC et aux opérations de migration ou de clonage, les destructions aux terminaisons des LRPC et les blocages aux attentes de résultats des LRPC. Bref, les opérations d'insertion ou de retrait des processus dans les files n'interviennent que lors de points de synchronisation bien particuliers de l'application.

En revanche, compte tenu du caractère préemptif de l'ordonnanceur de processus de la bibliothèque MARCEL, les opérations de commutation de contexte entre processus sont exécutées un très grand nombre de fois pendant la durée d'une application (typiquement une centaine de fois par seconde). Par conséquent, il apparaît clairement que l'ordonnanceur MARCEL doit assurer en priorité l'efficacité des opérations de changement de contexte, éventuellement au détriment de celles de l'insertion ou du retrait de processus dans la file des processus prêts.

Les opérations de commutation de contexte déclenchées par le mécanisme de préemption automatique (*i.e.* l'horloge) ont une durée d'exécution dépendant de deux facteurs. Le premier réside dans le temps nécessaire au système d'exploitation pour provoquer une interruption dans l'application. Sous UNIX, une telle interruption est réalisée par l'envoi d'un signal (`SIGALRM`) au processus lourd, ce qui a pour effet de dérouter son exécution vers une fonction spéciale de l'ordonnanceur MARCEL. L'efficacité de ce mécanisme dépend uniquement du système d'exploitation sous-jacent et est difficilement mesurable de manière précise au niveau utilisateur. Le second facteur réside dans l'efficacité de l'opération de commutation explicite exécutée lors d'une telle interruption. Cette opération, déclenchée par la fonction spéciale évoquée précédemment, est en revanche totalement sous le contrôle de l'ordonnanceur de l'application. C'est donc la primitive associée — `pthread_yield` — qui doit faire l'objet de toutes les attentions de MARCEL.

Le travail effectué par cette primitive se décompose en trois étapes :

1. le contexte du processus léger « courant » est sauvé dans son descripteur ;
2. un successeur lui est désigné et devient le processus courant ;
3. le contexte de ce dernier processus est restauré depuis son descripteur.

Les étapes (1) et (3) de cette commutation de contexte sont respectivement réalisées par des appels aux primitives `setjmp` et `longjmp` du langage C. Certaines bibliothèques de processus légers proposent des versions « assembleur » de ces routines [126], de façon à en accélérer l'exécution, mais au détriment d'une perte importante de portabilité.

Dans la sous-section suivante, nous montrons la stratégie d'ordonnancement adoptée dans la bibliothèque MARCEL pour garantir une exécution efficace de l'étape (2).

6.3.3.1 Principe de l'ordonnanceur

Nous l'avons vu au chapitre 5, l'ordonnancement des processus légers dans l'environnement PM^2 est un ordonnancement préemptif avec priorités. Plus précisément, cet ordonnanceur garantit que l'exécution des processus légers progresse, à un instant donné, à une vitesse proportionnelle à leur priorité divisée par le total des priorités des processus prêts dans le même module.

L'implantation de cet ordonnanceur repose sur un chaînage des processus légers prêts formant une liste circulaire ordonnée par priorités décroissantes (figure 6.7). La tête de cette liste est repérée par une variable globale que nous appellerons `tete`.

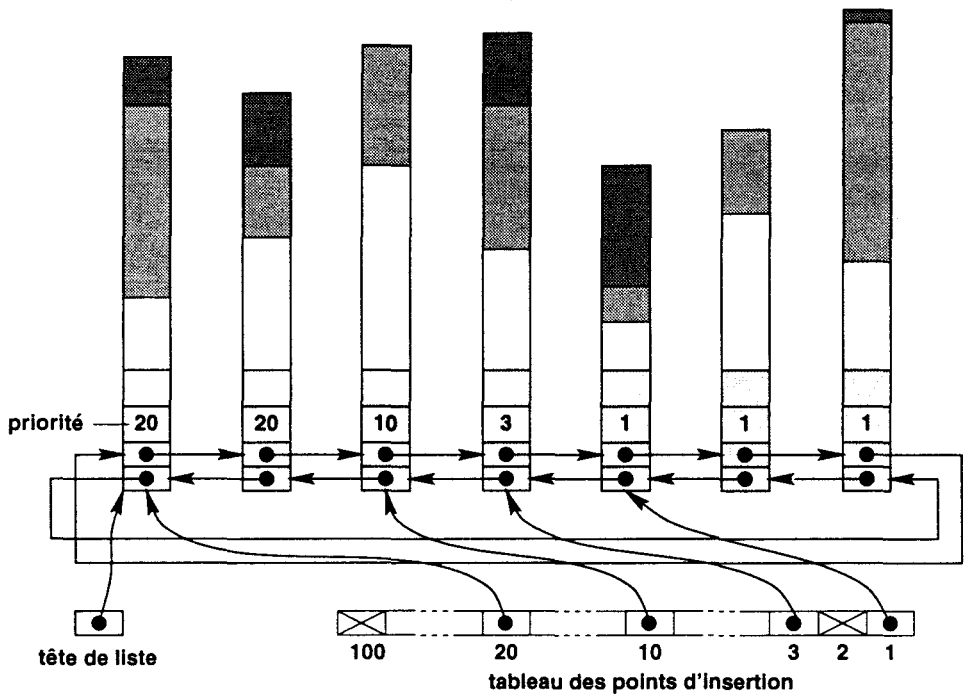

FIG. 6.7 - *Les processus prêts appartiennent à une liste circulaire doublement chaînée, ce qui permet aux opérations de désignation d'un successeur et de retrait de la liste d'être effectuées en temps constant. Grâce à une table de « points d'insertions », une insertion dans la liste est une opération en O(MAX_PRIO) dans le pire des cas.*

Chacun des processus légers contient, dans son descripteur, un champ (que nous appellerons `prio`) indiquant sa priorité et un champ (que nous appellerons `quanta`) contenant le nombre de quanta de temps accordés au processus léger depuis le début du *cycle d'ordonnancement*[20] courant. Lorsqu'un processus est inséré dans la file, le champ `quanta` est initialisé à la valeur de sa priorité.

20. Nous appelons *cycle d'ordonnancement* la période à l'issue de laquelle chacun des processus de la file a occupé le processeur pendant un nombre de quanta égal à sa priorité.

Lorsqu'une opération de commutation de contexte est déclenchée, le champ **quanta** du processus courant est décrémenté et sa valeur est comparée à celle du processus suivant. Si elle est inférieure, alors le processus suivant devient le processus courant. Sinon, c'est le processus désigné par la variable globale **tete** qui devient le processus courant. Notons que lorsque le champ **quanta** vaut zéro, la valeur du champ **prio** lui est automatiquement réaffectée. Ce fonctionnement est illustré en figure 6.8.

```
courant->quanta = courant->quanta-1;
if(courant->quanta == 0) {
   courant->quanta = courant->prio;
   courant = courant->suivant;
} else {
   if(courant->quanta >= courant->suivant->quanta)
      courant = tete;
   else
      courant = suivant;
}
```

FIG. 6.8 - *Algorithme de désignation du successeur d'un processus lors d'une commutation de contexte.*

Il est facile de voir que cet algorithme assure, sur une période de temps assez longue (*i.e.* sur un cycle d'ordonnancement), que chaque processus occupe le ratio du temps processeur qui lui est dû, c'est-à-dire sa priorité divisée par la somme totale des priorités des processus prêts. De plus, la désignation du successeur du processus courant s'effectue en **temps constant**, ne dépendant en aucune façon du nombre de processus prêts ni du nombre de niveaux de priorités gérés par l'ordonnanceur.

La contrepartie de l'efficacité de la primitive de commutation de contexte concerne l'opération d'insertion d'un nouveau processus dans la file, qui doit s'effectuer en conservant l'ordre des priorités. Pour conserver un temps d'exécution raisonnable, une table de « points d'insertion » indique, pour chaque classe de priorité, le premier processus de cette classe présent dans la file. Ainsi, dans le pire des cas, la recherche du point d'insertion dans la liste est une opération en O(MAX_PRIO). L'opération de retrait, quant à elle, est indépendante du nombre de niveaux de priorités.

6.3.3.2 Mesures

Afin d'évaluer l'ordre de grandeur d'une opération de commutation de contexte explicite avec la bibliothèque MARCEL, le tableau 6.2 indique les temps d'exécution de cette opération comparés avec les temps d'exécution de la paire d'instructions **setjmp/longjmp** sur quelques architectures répandues.

Afin d'évaluer, pour une application, le temps passé dans les mécanismes de commutation de contexte déclenchés par la préemption automatique, nous avons mesuré le temps d'exécution d'un programme contenant des processus légers effectuant un grand nombre d'itérations « à vide », et cela pour différentes fréquences de préemption (y compris sans préemption du tout). Le choix d'une application effectuant des itérations à vide est destiné à minimiser l'observation d'« effets de cache » entre les différentes mesures effectuées. Comme il était prévisible, la variation du nombre de processus légers impliqués dans l'application n'a pas du

TAB. 6.2 - *Temps de commutation de contexte (explicite) entre deux processus légers. Les temps d'exécution de la paire d'instructions* `setjmp/longjmp` *sont donnés pour indication.*

Architecture	Marcel	setjmp/longjmp
PC/Linux 120Mhz	0,969 μs	0,409 μs
Sparc 5/Solaris 85Mhz	8,838 μs	3,563 μs
Sparc ELC/SunOs	127,158 μs	86,643 μs

tout influencé les différences (en %) observées. Les résultats de cette expérience sont donnés dans le tableau 6.3.

TAB. 6.3 - *Influence de la fréquence de préemption sur le temps d'exécution d'un programme.*

	Sun Sparc5/Solaris 85MHz		PC/Linux 120MHz	
Période (μs)	Temps d'exécution	Surcoût	Temps d'exécution	Surcoût
sans préemption	13,024 s	—	5,009 s	—
10 000	13,475 s	3,5 %	5,018 s	0,1 %
20 000	13,251 s	1,7 %	5,014 s	0,1 %
50 000	13,118 s	0,7 %	5,011 s	< 0,1 %
100 000	13,072 s	0,3 %	5,010 s	< 0,1 %

Ces mesures montrent que, pour une période « standard[21] » de 20 millisecondes, la perturbation occasionnée par les commutations de contexte reste inférieure à 2 % du temps d'exécution de l'application. Compte tenu des nombreux avantages de la préemption évoqués au chapitre 5, nous considérons que cette valeur est acceptable. Si l'on compare les perturbations mesurées sur la machine Sun par rapport à celles mesurées sur PC, on remarque un écart bien plus important que ce que la différence de puissance des processeurs laissait présager. Cet écart semble dû à une gestion très coûteuse des signaux par le système Solaris. Sous Linux, par contre, la perturbation est si faible qu'il serait intéressant d'utiliser des fréquences de préemption plus rapides. Hélas, la version actuelle (2.0.22) du système ne le permet pas.

Notons enfin que ces résultats, puisqu'ils sont mesurés sur une application peu sensible aux effets de cache, doivent s'entendre « en moyenne ». En particulier, certaines applications subiront des perturbations plus importantes, alors que certaines pourront même afficher des performances supérieures à la version sans préemption. Nous reviendrons sur ce phénomène en conclusion de cette section.

6.3.4 Extension de pile

Au chapitre 3, nous avons évoqué un problème majeur relatif à l'utilisation des processus légers : le dimensionnement de leur taille de pile. Dans la plupart des bibliothèques de processus légers, cette taille est fixée à la création des processus et ne varie pas au cours de leur exécution, ce qui empêche l'exécution d'algorithmes récursifs dont on ne connaît pas la profondeur d'appel à l'avance. Dans le cadre du support des applications irrégulières, cette propriété est beaucoup plus importante qu'il n'y paraît. En effet, dans les applications développées avec des environnements tels que PM^2, DTS [20] ou Athapascan[34], le code exécuté

21. Fréquemment employée par défaut dans les bibliothèques de processus légers

par les processus légers est souvent jonché de « *points de choix* »[22] menant soit à une décomposition parallèle de l'algorithme (par un *appel de procédure distant*), soit à une exécution séquentielle de celui-ci, en fonction de critères évalués à l'exécution. Dans certains cas de figure, une telle approche peut mener certains processus à effectuer tous ces appels de façon séquentielle, voire récursive.

C'est pourquoi un mécanisme d'extension dynamique de pile est proposé dans l'environnement PM^2. Son utilisation reste malheureusement peu pratique, puisqu'il doit être effectué explicitement par les processus légers, au moyen de la primitive **pthread_growstack_np** qui prend en argument le nombre d'octets dont il faut augmenter la pile actuelle. Typiquement, une opération d'extension de pile doit être exécutée avant un appel de fonction risquant d'occasionner le débordement de cette dernière. Pour cela, une primitive **pthread_usablestack_np** peut être appelée par un processus pour consulter l'espace de pile restant. La figure 6.9 illustre cette utilisation.

```
/*
 * Avant d'effectuer un appel à la fonction f,
 * on vérifie la quantité de pile restante :
 */

if(pthread_usablestack_np() < 5000)
   pthread_growstack_np(5000);

f(...);
```

FIG. 6.9 - *Utilisation du mécanisme d'extension dynamique de pile.*

Dans cet exemple, la pile du processus est agrandie (de 5000 octets) si la quantité de pile restante est inférieure à 5000 octets. En réalité, ce seuil dépend de l'espace qui va être occupé par l'empilement du contexte de la fonction `f`, qui dépend principalement de la taille des données locales à cette fonction. L'estimation de cette taille peut s'effectuer soit par une estimation grossière (pour les fonctions dont le contexte local se compose de quelques variables de type simple) comme il a été fait dans l'exemple, soit par une préexécution de l'application mesurant les « consommations » de pile de chaque fonction du programme.

La réalisation de ce mécanisme repose sur l'extension de l'espace mémoire du processus (descripteur + pile + espace privé) réclamée au système par un appel à la primitive **realloc** de la librairie C. Une fois l'opération réalisée (ce qui peut avoir déplacé les données du processus en mémoire), la zone de pile « utilisée » est déplacée vers le haut, de façon à occuper toute la partie haute de ce nouvel espace. Dès lors, certaines adresses contenues dans la pile d'exécution ne sont plus valides : il faut effectuer une opération de « translation de pile », dont nous allons maintenant examiner le déroulement.

6.3.4.1 Mécanisme de base : translation de pile

D'un point de vue structurel, la pile d'exécution d'un processus léger est composée d'une superposition de blocs, représentant chacun le contexte d'exécution d'une fonction. Tout en haut de la pile se trouve le bloc correspondant à la fonction initialement exécutée par le

22. Tels que les *poly-algorithmes* d'Athapascan.

processus léger (passée en paramètre de `pthread_create`). Le bloc en bas de pile, quant à lui, représente le contexte de la fonction couramment exécutée par le processus. Entre ces deux blocs se trouve une liste de blocs représentant les appels intermédiaires entre ces deux fonctions. Généralement[23], tous ces blocs sont chaînés entre eux. Plus précisément, chaque bloc possède un pointeur sur le bloc immédiatement supérieur, qui, lors d'un retour de fonction, est utilisé pour repositionner le pointeur de bloc courant (*frame pointer*). Cette organisation est illustrée sur la figure 6.10.

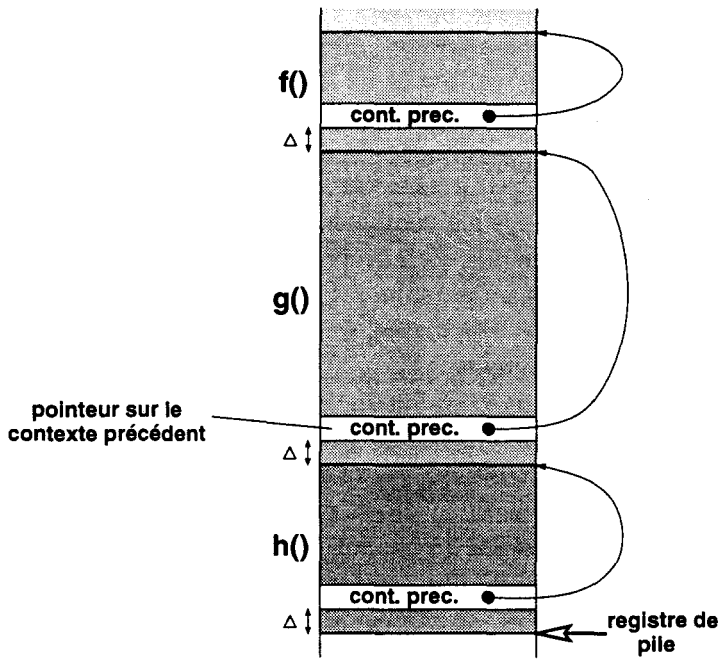

FIG. 6.10 - *La pile d'exécution d'un processus est un enchaînement de blocs (contextes) correspondant aux appels de fonctions en cours. Chaque nouveau bloc empilé mémorise l'adresse du précédent. Lors d'un déplacement de la pile en mémoire, ces pointeurs doivent être translatés, ainsi que la valeur du registre de pile.*

Lorsqu'une telle zone est déplacée en mémoire, les pointeurs constituant le chaînage ne contiennent évidemment plus des valeurs correctes. Pour les corriger, la bibliothèque MARCEL doit pouvoir parcourir cette liste de pointeur et donc connaître leur emplacement relativement au début de chaque contexte. Le problème réside dans le fait que cette information est dépendante du compilateur utilisé, voire même des options de compilation utilisées. Il n'est donc pas possible d'implanter un mécanisme de translation de pile qui fonctionnerait aveuglément quel que soit le compilateur utilisé.

C'est pourquoi nous avons choisi de *dépendre* du compilateur C de la fondation GNU, à savoir le compilateur `gcc`. Dans l'absolu, ce choix peut être considéré comme une limitation gênante. Dans la pratique, étant donné la large disponibilité de ce compilateur sur les architectures actuelles, nous n'avons pas encore été confronté au problème de sa non-disponibilité sur une machine particulière.[24]

23. Nous verrons plus tard qu'il est possible, sur certaines architectures, d'éviter ce chaînage.

24. MARCEL est actuellement opérationnel sur 6 architectures.

Grâce à cette dépendance, nous avons facilement pu réaliser le mécanisme de translation du chaînage des contextes évoqué précédemment. Cependant, cela ne constitue pas la seule tâche à effectuer lors d'une opération de translation de pile, car d'autres références (pointeurs) que ces « pointeurs de contexte » peuvent se trouver dans une pile d'exécution : les références liées aux traitants d'exception (cf. section 6.3.1.1) et les pointeurs déclarés au niveau applicatif.

La première catégorie de références est aisément corrigée par la bibliothèque MARCEL car les traitants d'exception sont chaînés entre eux. En revanche, la localisation des pointeurs de niveau applicatif ne peut pas, sans le contrôle de la phase de compilation, être automatiquement détectée par MARCEL. Il s'agit d'un problème déjà évoqué en section 5.4.5.3 à propos de la migration de processus, où nous avions vu qu'il était nécessaire à l'application de *déclarer* la localisation de ces pointeurs à l'aide de la primitive `pm2_register_pointer`. Le même mécanisme doit être utilisé dans le cas où un processus risque d'invoquer une opération d'extension de pile. L'appel de la primitive `pm2_register_pointer` a simplement pour effet de ranger l'adresse d'un pointeur dans une table contenue dans les données privées du processus. Lors d'une opération de translation de pile d'un processus, cette table est parcourue et la valeur de chacun des pointeurs enregistrés est examinée. Si ce pointeur contient l'adresse d'une donnée locale à la pile, alors l'adresse est translatée, sinon (pointeur NULL ou pointeur dans le tas) la valeur du pointeur reste inchangée.

En résumé, une opération d'extension de pile implique 1°) une opération d'allocation mémoire, 2°) un déplacement de la zone de pile utilisée et 3°) la translation des pointeurs de contextes, de la chaîne des traitants d'exception et des références de niveau applicatif. Notons que, sur certaines architectures, il est possible d'éviter (via l'option de compilation `-fomit-frame-pointer`) la présence du chaînage des contextes dans la pile d'exécution d'un processus, ce qui permet un gain de performance significatif.

6.3.4.2 Mesures

À titre indicatif, nous examinons ici les temps d'exécution de l'opération d'extension de pile en fonction du nombre de contextes contenus dans la pile (profondeur de récursivité) et de la taille de la pile. L'application choisie comporte un unique processus léger dont l'exécution effectue un très grand nombre d'appels récursifs (jusqu'à 3000) d'une fonction comportant peu de variables locales (pour favoriser la profondeur d'appel pour une taille de pile donnée). À chaque appel, le processus vérifie la taille de pile restante et, si elle est inférieure à 3 kilo-octets, l'augmente de cette même quantité. À chaque fois, nous avons mesuré la durée totale de l'opération d'extension, ainsi que la durée de l'opération de « reformatage de pile » (translation des pointeurs de contextes). Ces mesures ont été effectuées sur un PC 120Mhz sous Linux (sur une application compilée sans l'option `-fomit-frame-pointer`). Elles sont reportées dans le tableau 6.4.

Dans ce tableau, on peut vérifier que la durée de l'opération de reformatage de la pile est strictement proportionnelle au nombre de contextes contenus dans la pile. La différence entre la durée totale de l'opération et la durée du reformatage est directement liée aux opérations d'allocation et de déplacement de la pile en mémoire. En particulier, la durée de l'opération d'allocation est particulièrement imprévisible, ce qui explique l'étrange progression de la durée totale mesurée.

Notons enfin qu'en raison de la dégradation progressive des performances de l'opération d'extension au fur et à mesure que la profondeur d'appel augmente, il est préférable, par rapport à l'exemple étudié, d'accroître la taille de pile par morceaux plus importants, de

TAB. 6.4 - *Temps d'exécution de l'opération d'extension de pile en fonction de la profondeur d'appel de la fonction courante du processus léger.*

Profondeur d'appel	Taille de pile (Ko)	Temps du reformatage	Temps total
159	2 Ko	44 μs	349 μs
410	5 Ko	108 μs	207 μs
659	8 Ko	174 μs	414 μs
910	11 Ko	237 μs	518 μs
1159	14 Ko	300 μs	609 μs
1410	17 Ko	363 μs	579 μs
1659	20 Ko	424 μs	792 μs
1910	23 Ko	488 μs	879 μs
2159	26 Ko	555 μs	847 μs
2410	29 Ko	615 μs	1063 μs
2659	32 Ko	696 μs	1142 μs
2910	35 Ko	760 μs	1249 μs

façon à diminuer la fréquence de ces opérations. De ce point de vue, une valeur consistant à « doubler » la taille de pile lors de chaque extension nous semble être une bonne stratégie.

6.3.5 Hibernation et réveil

Outre l'extension de pile, les fonctionnalités véritablement « originales » fournies par l'environnement PM^2 en ce qui concerne la gestion des processus légers sont celles se rapportant aux opérations de migration, de clonage et, dans un futur proche, de va-et-vient sur disque. En fait, la réalisation de ces fonctionnalités repose principalement sur l'utilisation du mécanisme de *translation de pile* décrit dans la section précédente :

migration Une opération de migration consiste, schématiquement, à empaqueter l'image mémoire d'un processus dans un message, à transmettre ce message vers le nœud destinataire, à installer cette image à une adresse mémoire disponible et enfin à effectuer une translation de pile de manière à pouvoir redémarrer l'exécution du processus. Bien évidemment, le processus « original » est éliminé dès que son image est recopiée dans le message.

clonage L'opération de *séparation*[25], si elle possède une sémantique bien différente de l'opération de migration, n'en est pas moins sa « sœur jumelle » sur le plan de la réalisation. En effet, la séparation d'un processus en deux clones (pour n clones, il suffit de répéter l'opération) reprend les mêmes étapes qu'une migration, à ceci près que le processus original ne disparaît pas après la recopie de son image.

va-et-vient disque Le mécanisme de *va-et-vient* de processus légers entre la mémoire principale d'un nœud et la mémoire secondaire sera basé sur un mécanisme élémentaire (déjà disponible dans PM^2) de sauvegarde/chargement de processus sur disque. Là encore, la réalisation de ce mécanisme s'apparente à une opération de migration. En effet,

25. qui constitue la première étape d'une opération de clonage

en remplaçant, dans l'opération de migration, respectivement l'opération d'empaquetage et de transfert par une opération d'*écriture sur disque* et l'opération de réception et de désempaquetage par une opération de *lecture disque*, on obtient exactement un mécanisme de sauvegarde/chargement sur disque.

Compte tenu des liens de parenté étroits qui existent entre ces trois mécanismes, **un seul mécanisme « générique »** de base a été implanté dans la bibliothèque MARCEL. Ce mécanisme, appelé « *hibernation de processus* », comporte deux phases :

hibernation La bibliothèque MARCEL fournit une primitive permettant de figer un processus léger (`pthread_begin_hibernation_np`) et d'en obtenir un « bloc de glace », qui est tout simplement une suite d'octets caractérisant le processus. Cette fonction attend bien sûr l'identificateur du processus en paramètre, mais aussi un booléen spécifiant si le bloc de glace contient le processus lui-même (auquel cas il ne survivra pas sous sa forme originelle) ou bien une copie du processus (un clone glacé en quelque sorte[26]). Enfin, l'adresse d'une fonction particulière doit également être passée en paramètre. Cette fonction sera appelée depuis la primitive `pthread_begin_hibernation_np` et recevra le bloc de glace résultant en paramètre (par adresse). C'est dans cette fonction qu'il sera possible d'envoyer le bloc de glace sur un réseau de communication, ou encore de le sauver dans un fichier. Une fois l'exécution de cette fonction terminée, le contrôle retourne à la primitive `pthread_begin_hibernation_np` qui éliminera le processus original si cela a été stipulé.

réveil De façon symétrique, la bibliothèque MARCEL fournit une primitive permettant de réveiller un processus contenu dans un bloc de glace (`pthread_end_hibernation_np`). De manière interne, cette primitive repose principalement sur l'utilisation du mécanisme de *translation de pile* décrit dans une section précédente. Accessoirement, cette primitive accepte une fonction en argument qui est appelée entre le reformatage de pile du processus et son réveil effectif. Elle permet en outre d'effectuer des traitements post-migration comme par exemple la réallocation de données dynamiques qui auraient été conservées avec le bloc de glace.

C'est donc à l'aide de ces deux mécanismes principaux que sont implantées, comme nous le verrons dans les sections ultérieures, les fonctionnalités de migration, de clonage et de sauvegarde sur disque de processus.

Pour des raisons liées à l'implantation actuelle, le mécanisme d'*hibernation* de processus ne peut s'appliquer qu'à des processus *actifs*. Autrement dit, l'appel de la primitive `pthread_begin_hibernation_np` sur un processus léger bloqué sur une primitive de synchronisation n'est pas autorisé.[27] Pour l'instant, cette « lacune » est sans conséquence pour les applications utilisant PM^2 car de toute façon les opérations de migration et de clonage ne peuvent pas être appelées sur de tels processus[28]. En revanche, elle n'est pas acceptable dans la perspective de conception d'un mécanisme de va-et-vient de processus dans l'environnement PM^2, car ce dernier mécanisme nécessiterait avant toute chose de pouvoir « migrer

26. désolé...

27. Principalement parce que les problèmes techniques posés par l'implantation efficace d'une telle fonctionnalité sont très ardus...

28. Dans le cas du clonage, c'est évident puisque le processus s'« auto-congèle ». Dans le cas de la migration, l'environnement l'assure en ne donnant que la liste des processus actifs lors d'un appel à `pm2_threads_list`.

sur disque » des processus bloqués. C'est pourquoi, parallèlement aux travaux de recherche qui seront prochainement démarrés sur l'intégration du mécanisme de va-et-vient dans PM^2, des travaux d'implantation seront menés pour combler cette lacune.

6.3.6 Comparaison avec d'autres bibliothèques

Afin de vérifier la réelle efficacité des choix de conception de la bibliothèque MARCEL, nous avons effectué des tests de performance comparatifs avec un certain nombre de bibliothèques présentant une interface POSIX. Il s'agit des bibliothèques *Pthreads* de Frank Mueller [127], de Chris Provenzano [143] et du système Solaris[29] [157]. Pour chacune d'elles, nous avons mesuré le temps d'exécution d'une opération de changement de contexte (moyenne sur 1 000 commutations), d'une création de processus[30] (moyenne sur 100 créations successives) et d'une application développant un arbre de 1 000 processus légers pour effectuer un traitement dichotomique (en l'occurrence la somme des entiers de 1 à 500). Cette dernière application, qui sollicite à la fois les opérations création/destruction de processus mais aussi celles de synchronisation, exécute l'algorithme par deux fois, ce qui permet d'observer, la deuxième fois (*Arbre bis*), une éventuelle accélération due à la réutilisation de caches de piles ou autres. Les mesures, rigoureusement effectuées à partir des mêmes programmes[31], sont présentées dans le tableau 6.5.

TAB. 6.5 - *Comparaison de la bibliothèque* MARCEL *avec les principales bibliothèques (POSIX) existantes.*

Sparc5/Solaris 85Mhz				
Opération	MARCEL	Mueller	Provenzano	Solaris
Chgt contexte	8,838 μs	220,232 μs	34,366 μs	48,196 μs
Création	37,630 μs	45,193 μs	26,791 μs	86,838 μs
Création/Terminaison	85,252 μs	532,201 μs	109,161 μs	305,937 μs
Arbre	1,069 s	2,472 s	2,214 s	1,978 s
Arbre bis	0,110 s	1,489 s	0,434 s	1,947 s

Dans ce tableau, on remarque que les performances de la bibliothèque MARCEL sont à la hauteur des espérances évoquées précédemment. L'opération de commutation de contexte, par exemple, confirme son bon comportement dû à une prise de décision rapide de l'ordonnanceur pour déterminer le « processus successeur » du processus courant. La version de Frank Mueller exhibe de pauvres performances dues au surcoût d'appels à des primitives du noyau UNIX pour la gestion des signaux.

Les résultats concernant les opérations de création de processus, quant à eux, sont à interpréter avec prudence. En effet, il est immédiat de constater l'existence d'écarts de temps d'exécution importants, pour certaines bibliothèques, entre une opération de création asynchrone de processus et la même opération suivie de l'attente de sa terminaison (le code applicatif exécuté par le processus étant vide). Outre au surcoût « normal » occasionné par l'instruction d'attente de terminaison, ces écarts sont principalement dus à l'implantation

29. Précisons qu'il s'agit des processus légers de *niveau utilisateur* de Solaris.

30. Deux cas ont été mesurés : la création asynchrone et le cycle création-exécution d'une fonction vide-terminaison (`pthread_create/pthread_join`).

31. En particulier, les tailles de piles utilisées étaient identiques.

de l'opération de création qui repousse l'exécution d'un certain nombre de traitements au moment où le processus obtiendra le processeur pour la première fois.

De ce fait, le deuxième type de mesures (*création/terminaison*) s'avère être beaucoup plus significatif que le premier. En examinant ces dernières, il apparaît que deux des quatre bibliothèques testées présentent des opérations d'allocation de processus particulièrement coûteuses : il s'agit de celles de Mueller et de Solaris. En fait, cette différence provient du fait que ces deux bibliothèques fournissent des fonctionnalités supplémentaires aux deux précédentes, telles que des mécanismes de test de débordement de pile[32] ou des mécanismes de gestion des signaux UNIX. Compte tenu de l'intérêt somme toute limité de ce type de fonctionnalités pour le support des applications irrégulières, ces résultats montrent l'importance d'utiliser une gestion des processus légers « *sur mesure* » dans un cadre d'utilisation tel que celui du projet ESPACE.

L'observation des différences de temps d'exécution entre les deux lancements du calcul dichotomique (créant un processus par nœud de l'arbre) fait apparaître (sauf dans le cas de Solaris) un phénomène d'accélération dû à l'utilisation de caches de pile pour les bibliothèques MARCEL et Provenzano et simplement dû aux primitives d'allocation mémoire (`malloc`) du système pour la bibliothèque Mueller.

Si les résultats présentés ici montrent parfois des différences importantes entre les bibliothèques, il convient néanmoins de les relativiser dans un cadre d'utilisation appliqué aux applications de calcul intensif. En effet, certains de ces écarts s'estomperont par exemple avec des applications ne sollicitant pas beaucoup d'opérations de création dynamiques (rendant les mécanismes de cache de pile inutiles) ou encore avec des applications n'utilisant tout simplement qu'un petit nombre de processus légers. En revanche, une caractéristique telle que l'efficacité de l'opération de commutation de contexte gardera toujours son importance dans le cas d'un ordonnancement préemptif, quelle que soit l'application visée.

6.3.7 Conclusion sur Marcel

Dans cette section, nous avons décrit les principales caractéristiques de MARCEL, une bibliothèque de processus légers de niveau utilisateur conçue dans le cadre de cette thèse.

Nous avons montré comment, sous forme d'extensions à l'interface POSIX-threads, il est possible de bâtir une bibliothèque fournissant des fonctionnalités évoluées (exceptions, hibernation, etc.) tout en conservant un degré de portabilité élevé. À l'heure actuelle, la bibliothèque MARCEL est opérationnelle sur six architectures : Sparc/SunOS, Sparc/Solaris, Alpha/OSF-1, PC/Linux, PowerPC/Aix et Mips/Irix. Le portage de cette bibliothèque sur une nouvelle architecture[33] nécessite très peu de travail. Principalement, il s'agit de l'écriture de quelques instructions « en assembleur » permettant de positionner le registre de pile à une valeur donnée ainsi que du calibrage du mécanisme de translation de pile[34]. Les sources de cette bibliothèque sont téléchargeables à partir de l'adresse `http://www.lifl.fr/~namyst/pm2.html`.

Cette bibliothèque étant, contrairement aux « bibliothèques classiques », spécialement conçue pour constituer la base d'un environnement distribué — en l'occurrence PM^2 — destiné au support des applications irrégulières parallèles, un effort particulier a été effectué pour favoriser l'efficacité d'exécution de ces applications. Dans ce cadre, nous avons évoqué, sur des points précis tels que l'allocation des processus ou la commutation de contexte, les

32. par verrouillage du bas de la pile, ce qui n'est pas fiable à 100 %
33. Et par la-même celui de PM^2
34. c'est-à-dire de déterminer l'emplacement du pointeur de bloc précédent au sein d'un bloc.

optimisations qu'il était possible d'effectuer pour augmenter les performances des applications. Dans le même esprit, nous avons décidé de ne pas implanter directement[35] un mécanisme de gestion des signaux « à la UNIX », à cause du surcoût qu'il occasionnerait sur les primitives de création de processus et de commutation de contexte. La comparaison des performances de MARCEL avec d'autres bibliothèques de processus léger (de type POSIX) confirme ces propriétés d'efficacité et montre en outre que l'intégration de mécanismes tels que l'hibernation de processus, qui constitue la base du mécanismes de migration de PM^2, peut s'effectuer sans incidence néfaste sur les performances générales d'une bibliothèque de processus légers.

Pour compléter les évaluations précédentes, nous allons terminer cette section en examinant l'influence[36] que peut avoir l'utilisation de processus légers dans une application parfaitement régulière sur une machine monoprocesseur. En fait, il s'agit principalement de mesurer la variation du temps d'exécution occasionnée par l'introduction de processus légers dans une application initialement séquentielle, pour laquelle un ordonnancement préemptif gérant de nombreux processus actifs risque, par exemple, de perturber les caches internes du processeur.

Le tableau 6.6 indique les temps d'exécution d'une multiplication de deux matrices d'entiers (512×512) en fonction du nombre de processus légers utilisés (découpage par blocs carrés), comparés au temps de référence établi par une version purement séquentielle. Cette expérience a été menée avec deux bibliothèques préemptives (quanta de 20 millisecondes) qui sont MARCEL et Mueller's Pthreads.

TAB. 6.6 - *Influence du nombre de processus légers utilisés pour la multiplication de deux matrices* 512×512.

Sun Sparc5/Solaris 85MHz				
Nombre de threads	MARCEL		Mueller's Pthreads	
	Temps d'exécution	Surcoût	Temps d'exécution	Surcoût
aucun	178,236 s	—	178,236 s	—
1	177,890 s	−0,2%	178,928 s	0,3%
4	178,326 s	0%	178,096 s	0%
16	178,276 s	0%	179,309 s	0,6%
64	178,129 s	0%	179,377 s	0,6%
256	178,273 s	0%	180,103 s	1%
1024	178,681 s	0,2%	181,136 s	1,6%

Comme on peut le constater, ces résultats sont très positifs, car ils montrent que le surcoût de la version parallèle contenant 1 024 processus légers est seulement de 0,2% avec la bibliothèque MARCEL (1,6% dans le cas de Mueller), ce qui montre la faible perturbation « en moyenne » occasionnée par l'ordonnancement préemptif des processus légers sur les performances d'une application.

Certains de ces résultats vont même à l'encontre de l'intuition, puisque les performances de la version « avec processus légers » sont meilleures que celles de la version séquentielle (MARCEL 1 et 64, Mueller 4). Dans [139], Christian Perez (LIP) observe le même phénomène dans des conditions similaires (application multiprogrammée avec MARCEL manipulant des

35. Par contre, la bibliothèque MARCEL fournit un mécanisme de « déviation de processus » permettant à une application de développer une gestion des signaux UNIX.

36. Intuitivement défavorable...

tableaux) et l'explique par des effets de cache du processeur favorisés par la « localité » des calculs effectués par chaque processus léger.

Abstraction faite des effets de cache, la faible augmentation de la perturbation par rapport au nombre de processus légers s'explique par un surcoût uniquement dû (du moins dans le cas de MARCEL) aux opérations de création et de destruction de processus.[37] Dans un cadre où l'utilisation des processus légers a pour but de virtualiser l'architecture sous-jacente, ces résultats se révèlent particulièrement encourageants. Dans le chapitre 7, nous examinerons une application d'optimisation combinatoire développée avec PM² dont les performances, « malgré » l'utilisation d'un très grand nombre de processus légers concurrents, sont en accord avec ces résultats.

6.4 Réalisation de PM²

La réalisation de l'environnement PM², comme indiqué en introduction de ce chapitre, repose sur les bibliothèques MARCEL et PVM, dont nous venons d'examiner les principales caractéristiques. Nous ne présenterons ici que les très grandes lignes de cette réalisation, en laissant de côté certaines « caractéristiques annexes »[38] au profit des mécanismes directement liés aux fonctionnalités principales de PM² que sont l'*appel de procédure à distance* et la *migration de processus*. Nous indiquerons également l'état d'avancement de l'implantation du mécanisme de *clonage léger* et les travaux restant à faire dans ce domaine.

6.4.1 PVM et processus légers

Comme nous l'avons évoqué au chapitre 3 (section 3.5.2), l'utilisation conjointe d'une bibliothèque non « *thread-safe* » (telle que PVM) et d'une bibliothèque de processus légers pose problème en raison de la non-réentrance de certaines primitives. L'exemple qui avait été évoqué dans cette section concernait l'utilisation concurrente des primitives `pvm_recv` et `pvm_send`, et mettait en évidence le problème posé par l'aspect « bloquant » de la primitive `pvm_recv`, empêchant ainsi sa protection par un mécanisme de verrou.

En fait, la non-réentrance de PVM ne constitue pas le seul problème posé par son utilisation dans des programmes multiprogrammés au niveau utilisateur. En effet, la bibliothèque MARCEL, comme toutes les bibliothèques de niveau utilisateur portables sur un grand nombre d'architectures, ne gère pas les appels bloquants au système d'exploitation (cf. section 6.3.1). Par conséquent, même si la bibliothèque PVM était réentrante, il ne serait pas possible d'autoriser l'exécution de la primitive `pvm_recv` par un processus léger sans s'assurer que le message attendu soit disponible (pour éviter tout blocage).

Ce problème, bien connu des concepteurs d'environnements à base de processus légers, oblige donc l'adoption d'une politique de *scrutation non-bloquante* périodique du réseau de communication lorsqu'il s'agit d'attendre l'arrivée d'un message. L'environnement PVM fournit une primitive — `pvm_nrecv` — permettant d'effectuer une telle scrutation.

D'un point de vue historique, nos premières expérimentations concernant l'intégration de processus légers avec PVM ont été menées dans le projet PVC [135]. À l'époque, nous souhaitions 1°) diminuer le coût de la scrutation occasionné par `pvm_nrecv` et 2°) rendre

37. Rappelons à cet égard que la perturbation due aux commutations de contexte ne dépend pas du nombre de processus légers employés (pour une quantité globale de travail fixée).

38. Le lecteur intéressé pourra se reporter aux nombreux exemples fournis avec la distribution standard à l'adresse `http://www.lifl.fr/~namyst/pm2.html`.

l'utilisation de PVM la plus transparente possible pour les différentes couches logicielles du projet. C'est pourquoi nous avons réalisé une version modifiée de la bibliothèque PVM offrant une interface réentrante et « *thread-synchrone* » vis-à-vis de la bibliothèque MARCEL. Dans cette version modifiée, les accès aux variables globales (tampons de communication, etc.) sont protégés par des verrous d'exclusion mutuelle et la primitive `pvm_recv` a été modifiée de façon à effectuer une scrutation interne à l'aide de la primitive `tselect` fournie par MARCEL. L'avantage de cette approche, outre d'autoriser la concurrence des appels aux primitives `pvm_send` et `pvm_recv`, réside dans la diminution du coût de scrutation du réseau car un appel à la primitive `tselect` est presque deux fois plus rapide qu'un appel à la primitive `pvm_nrecv`. Dans la section suivante, nous évaluerons le gain obtenu par cette modification.

Malheureusement, l'utilisation d'une version modifiée de PVM possède deux inconvénients majeurs. Le premier concerne la très faible pérennité de l'investissement réalisé, puisqu'à chaque évolution de PVM (qui continue d'évoluer encore), il faut recommencer ces modifications. Le deuxième inconvénient est encore plus gênant, car il empêche le passage d'une application utilisant cette version sécurisée sur une version propriétaire de PVM (par exemple PVMe sur l'IBM SP2) dont la mécanique interne (*i.e.* les sources) demeure inaccessible.

Par conséquent, la réalisation de l'environnement PM^2 a été effectuée de manière à permettre soit l'utilisation de notre version sécurisée de PVM (sur les machines ne disposant pas de versions propriétaires), soit l'utilisation d'une version non-sécurisée de PVM (auquel cas toutes les précautions d'utilisation sont prises par l'environnement PM^2 lui-même). Dans la section suivante, nous comparons plus précisément ces deux approches.

6.4.2 Traitement des requêtes

Dans le modèle de programmation PM^2, nous avons vu qu'une application était constituée d'un ensemble de *modules* contenant chacun une partie des processus légers de l'application. En fait, dans l'implantation actuelle, l'exécution de chacun des modules est prise en charge par une tâche PVM. Les identifiants de modules manipulés au niveau applicatif sont d'ailleurs directement des identifiants de tâches PVM (`tid`). En ce qui concerne la mise en place des modules sur les différents nœuds de l'architecture, nous avons également conservé la philosophie PVM en fournissant une primitive `pm2_spawn` ayant un profil et une sémantique identique à la primitive `pvm_spawn` sur laquelle elle s'appuie. Typiquement, le lancement d'une application PM^2 s'effectue donc de la même manière qu'une application PVM : on lance un premier module « manuellement » qui va ensuite effectuer des `pm2_spawn` pour exploiter les divers nœuds de l'architecture. À l'issue de ces créations de modules, le module initial peut alors, au moyen de LRPC, communiquer avec les autres modules et leur transmettre la liste complète des modules de l'application.

On le devine aisément, les mécanismes de LRPC, de migration et de clonage vont tous impliquer l'envoi de messages entre les modules. Si cela ne pose pas de problème particulier du côté du processus « initiateur de l'opération » (puisque c'est lui qui va exécuter l'opération `pvm_send`), il n'en est pas de même pour l'opération de réception (et donc de scrutation) qui doit être prise en charge de façon transparente par l'environnement. Plus précisément, le problème consiste à définir une *stratégie de scrutation périodique* du réseau réalisant un bon compromis entre une bonne réactivité des modules (favorisée par une scrutation fréquente) et une faible perturbation occasionnée par les scrutations inutiles (diminuée par une scrutation occasionnelle). Cette problématique a déjà été abondamment traitée par la communauté et le lecteur intéressé pourra par exemple se rapporter aux documents [73] (Nexus), [31] (Athapas-

can) ou [91] (Chant). En fait, il s'avère que l'adéquation de chacune des solutions possibles (scrutation par un processus spécial, scrutation à chaque commutation de contexte, etc.) dépend du type d'ordonnancement en vigueur (préemptif ou pas) et surtout du comportement des applications visées (ratio communications/calculs).

Dans l'environnement PM², nous avons choisi d'utiliser un processus léger spécial (dans chaque module) dédié à la réception et au traitement des requêtes provenant des autres modules (figure 6.11). Ce processus léger (créé lors de l'exécution de `pm2_init()`) possède la plus haute priorité possible, ce qui permet d'assurer des délais inter-scrutation bornés par le nombre de processus prêts multiplié par la durée d'un quantum de temps élémentaire.

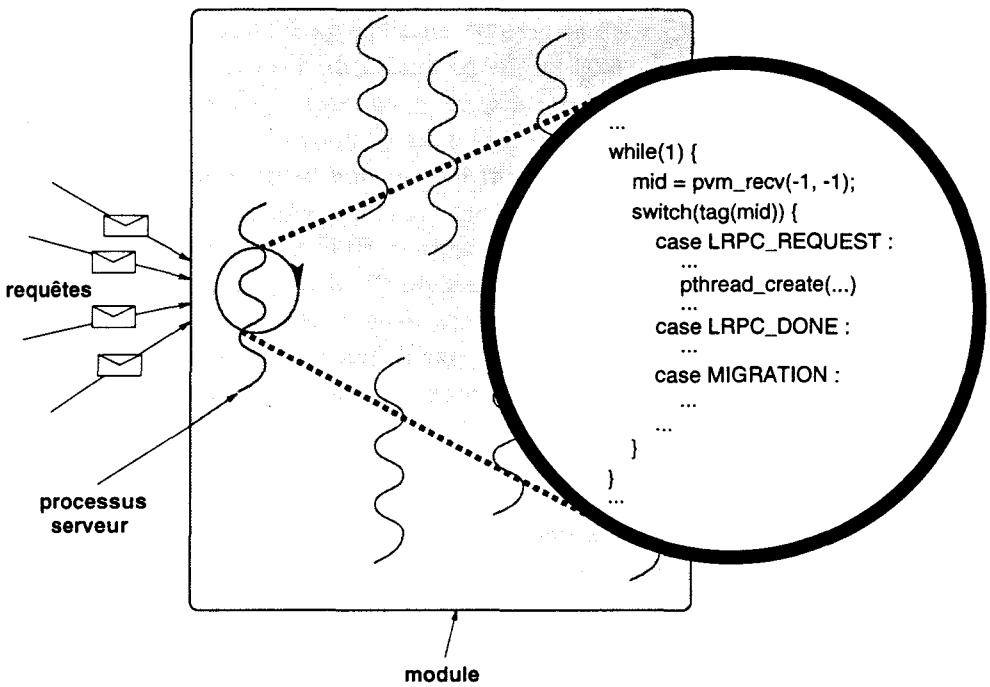

FIG. 6.11 - *Dans chaque module, un processus léger spécial — le « processus serveur » — est chargé de la réception et du traitement des messages en provenance de l'extérieur.*

Ce choix représente, à nos yeux, un très bon compromis pour les raisons suivantes :

- Une scrutation du réseau « à chaque commutation de processus » est beaucoup trop coûteuse dans le cadre du support d'applications parallèles de type scientifique, où un nouveau message de requête n'arrive pas toutes les 10 ou 20 millisecondes (période de préemption) sur chacun des modules.

- La période maximale pouvant s'écouler entre deux scrutations est proportionnelle au nombre de processus prêts dans le module. Cette réactivité variable des modules ne dépend cependant pas de la durée d'exécution de ces processus, mais diminue uniquement lorsque le degré de parallélisme de l'application augmente, ce qui est une caractéristique somme toute acceptable.

- Cette implantation est très bien adaptée au traitement des requêtes d'exécution de services en mode *quick*, puisque le processus « serveur » peut avantageusement remplir

cette fonction, en évitant ainsi des commutations de contexte inutiles vers un autre processus et en sérialisant ces requêtes de façon naturelle.

Comme on le voit sur la figure 6.11, le code exécuté par ce processus *serveur de requêtes* consiste en l'exécution en boucle infinie[39] d'une réception de message suivie de son traitement en fonction de son `tag` (utilisé pour *typer* les requêtes). Le code indiqué sur la figure suppose l'utilisation sous-jacente de la version sécurisée de PVM, ce qui signifie que l'appel à `pvm_recv` équivaut à :

```
faire {
   tselect(descripteurs_internes_de_pvm);
   si tselect_a_echoué alors
      /* passer la main au suivant : */
      pthread_yield();
} tant_que tselect_a_echoué;
lire_les_données_et_les_placer_dans_le_tampon_de_réception;
```

Dans le cas où une version non-modifiée de PVM est utilisée, le code de devient :

```
faire {
   pvm_nrecv(-1, -1);
   si pvm_nrecv_a_echoué alors
      /* passer la main au suivant : */
      pthread_yield();
} tant_que pvm_nrecv_a_echoué;
pvm_recv(-1, -1);
```

Hormis pour les machines disposant d'une version propriétaire de PVM[40], nous nous sommes posé la question de savoir si le gain de performance obtenu en utilisant la version sécurisée justifiait encore sa maintenance. Pour cela, nous avons mesuré la « perturbation maximale » occasionnée par les scrutations du *processus serveur* sur le déroulement d'un programme dans le cas le plus défavorable, c'est-à-dire celui où il n'y a que deux processus légers dans le module (processus serveur + processus effectuant le traitement). Nous avons donc mesuré le temps d'exécution séquentiel (en dehors de PM²) d'un traitement (constitué d'une grande boucle itération à vide), puis nous avons mesuré son temps de traitement lorsqu'il se trouve en présence du *processus serveur* de PM² pour chacune des deux versions. Les expériences ont été menées sur une station Sparc5 sous Solaris et sur un Pentium sous Linux, avec une période de préemption de 20 millisecondes dans les deux cas. Ces mesures sont récapitulées dans le tableau 6.7.

Comme on peut le constater, l'écart entre les deux versions existe, mais il n'est pas flagrant (0,6% sur Sparc5, 0,1% sur Pentium120). Notons tout de même que cette différence s'accentuerait s'il était possible de spécifier des périodes de préemption beaucoup plus rapides que 10 millisecondes, qui représente une valeur trop élevée pour des processeurs cadencés à plus de 100 méga-hertz. Il apparaît néanmoins que le faible écart de perturbation ne plaide pas en faveur de la version sécurisée de PVM, que nous avons par conséquent décidé de ne plus maintenir à jour.

La stratégie de scrutation étant fixée, nous allons maintenant brièvement examiner le déroulement des opérations effectuées lors d'*un appel de procédure à distance léger*.

39. En fait, elle sera interrompue lorsque l'application notifiera sa terminaison.

40. Auquel cas il est fortement conseillé d'utiliser cette version !

TAB. 6.7 - *Influence de la scrutation périodique sur le temps d'exécution d'un programme.*

Primitive de scrutation	Sun Sparc5/Solaris 85MHz		PC/Linux 120MHz	
	Temps d'exécution	Surcoût	Temps d'exécution	Surcoût
sans scrutation	13,070 s	—	10,012 s	—
`tselect`	13,340 s	2%	10,042 s	0,3%
`pvm_nrecv`	13,420 s	2,6%	10,052 s	0,4%

6.4.3 Appels de procédure distants

Afin de dresser les principales caractéristiques de l'implantation du mécanisme d'*appel de procédure à distance léger*, nous allons examiner les différents mécanismes impliqués lors d'un appel à attente différée (le fonctionnement des autres modes pouvant facilement en être déduit).

Un appel de procédure à distance se déroule en trois grandes phases : l'appel, le traitement de la requête et le retour de résultat.

6.4.3.1 Appel

Lorsqu'un processus léger effectue un appel à la primitive `LRPC_CALL(module, ...)`, un premier test est effectué pour vérifier qu'il ne s'agit pas d'un appel *local*. Si c'est le cas, alors le processus appelant crée un processus léger (`pthread_create`) pour prendre en charge le traitement associé au service, comme il est décrit dans la section suivante. La plupart du temps, l'appel n'est pas local et le processus doit donc envoyer une *requête de traitement distant* au module cible.

Tout d'abord, le processus va initialiser la structure d'attente (`pm2_wait_struct`) passée en paramètre qui servira à synchroniser l'attente des résultats avec leur arrivée effective. Cette structure contient donc essentiellement un sémaphore et l'adresse mémoire où il faudra ranger les résultats dès leur arrivée. Dans le cas où le processus appelant a spécifié qu'il désirait rester migrable entre l'appel et l'attente (`pm2_keep_migratable(TRUE)`), alors une structure d'attente fixe est allouée dans la mémoire globale du module et cette information est mémorisée dans la structure d'attente qui appartient au contexte local du processus.

Ensuite, un nouveau tampon de communication PVM est créé. Le choix de l'encodage à utiliser est effectué en examinant une table interne pour tenter de déterminer le type d'architecture sur lequel s'exécute le module cible. Cette table doit être mise à jour explicitement par l'utilisateur (car l'opération est coûteuse) par un appel à `pm2_update_config`. Si la recherche échoue, ou si l'architecture ne correspond pas à l'architecture locale, l'encodage XDR est utilisé.

Une fois le tampon créé, les différents paramètres du LRPC (numéro de service, priorité, taille de pile) sont empaquetés, sans oublier les arguments du service (qui sont empaquetés en appelant la fonction souche correspondante écrite par le programmeur) et un pointeur sur la structure d'attente. Le message est enfin envoyé vers le module cible (`pvm_send`) avec le `tag` « `LRPC_REQUEST` ».

6.4.3.2 Traitement

Lors de la réception d'un message de type `LRPC_REQUEST` par un processus serveur, les données telles que le numéro du service à exécuter, la priorité du processus à créer ainsi que sa taille de pile sont extraites du message. Un nouveau processus léger est ensuite créé (`pthread_create`) en fonction des caractéristiques spécifiées. Notons que si la taille des arguments et des résultats du service est supérieure à une constante fixée, alors la taille de pile du processus est augmentée de ces quantités. Une fois créé, le processus peut alors recevoir ses arguments (via une fonction souche de désempaquetage) et son exécution est lancée sur une fonction intermédiaire.

Le principal rôle de cette fonction intermédiaire est de rattraper les éventuelles exceptions levées durant l'exécution du service, pour les propager au processus appelant (`tag LRPC_FAILED`). Si tout se passe bien, la terminaison du service déclenche un envoi de message vers le site du processus appelant contenant les résultats du traitement (`tag LRPC_DONE`).

Notons que la migration du processus léger exécutant un service n'a aucune incidence sur le déroulement des opérations que nous venons de décrire.

6.4.3.3 Retour du résultat

Lorsqu'un processus exécute la primitive `LRP_WAIT`, deux cas peuvent se produire :

1. Le processus a migré entre temps. Dans ce cas, il envoie un message sur le site d'appel pour signaler où il est, puis il se met en attente sur le sémaphore de sa structure d'attente locale.

2. Le processus n'a pas migré (ou bien il est revenu sur le même site qu'au moment de l'appel). Dans ce cas, il se bloque simplement sur le sémaphore de la structure d'attente.

Lors de la réception d'un message de type `LRPC_DONE` par un processus serveur, deux cas peuvent à nouveau se produire :

1. Au moment de l'appel, le processus appelant avait manifesté son désir de rester migrable. Dans ce cas, soit il est déjà en attente du résultat, auquel cas le système sait où il est et peut lui « faire suivre » les résultats ; soit il n'est pas encore en attente du résultat, auquel cas le message contenu dans le tampon courant en réception est sauvé et « mis de côté » pour le faire suivre ultérieurement (ou éventuellement le désempaqueter localement).

2. Au moment de l'appel, le processus a promis qu'il ne migrerait pas. Dans ce cas, les résultats sont extraits du message à l'aide de la quatrième et dernière fonction souche à l'emplacement désigné par un des champs de la structure d'attente et le sémaphore est signalé, ce qui débloque — ou ne bloquera pas — le processus appelant.

Lorsque le processus bloqué sur le sémaphore est réveillé, l'exécution de la primitive `LRP_WAIT` est terminée et le processus peut enfin consulter les résultats de l'appel. Notons que si c'est un message de type `LRPC_FAILED` qui est reçu, les étapes précédentes sont à peu près identiques, sauf que l'exécution de la primitive `LRP_WAIT` lève l'exception `LRPC_ERROR` pour le processus appelant.

6.4.3.4 Mesures

Afin d'évaluer les performances du mécanisme d'appel de procédure à distance de l'environnement PM^2, notamment en comparaison avec les performances de la librairie de communication native utilisée (PVM), nous avons mesuré le temps d'exécution d'un appel de chacun des quatre modes de LRPC, pour différentes tailles de données (arguments et résultats). Dans un premier temps, nous comparons les primitives d'appel asynchrone (avec et sans création de processus légers) avec l'envoi de message PVM. Les programmes de test effectuent tous un « *ping-pong* » entre deux modules impliquant 1 000 envois de messages au total. Pour les deux programmes PM^2, le code des services exécutés de part et d'autre consiste à re-solliciter un LRPC vers le module appelant, ce qui mène à un ping-pong jusqu'à ce que le nombre de LRPC voulu soit atteint. Lorsque cette condition est atteinte, le contrôle est revenu dans le module de départ (nombre pair de LRPC) ce qui termine la mesure (divisée par 1 000 pour la ramener au cas d'un seul appel).

Les mesures ont été effectuées entre deux modules situés sur la même machine (Pentium120) de manière à minimiser les temps de transferts « extérieurs » et maximiser le surcoût relatif de PM^2 par rapport à PVM. Les résultats de cette expérience sont montrés sur la figure 6.12.

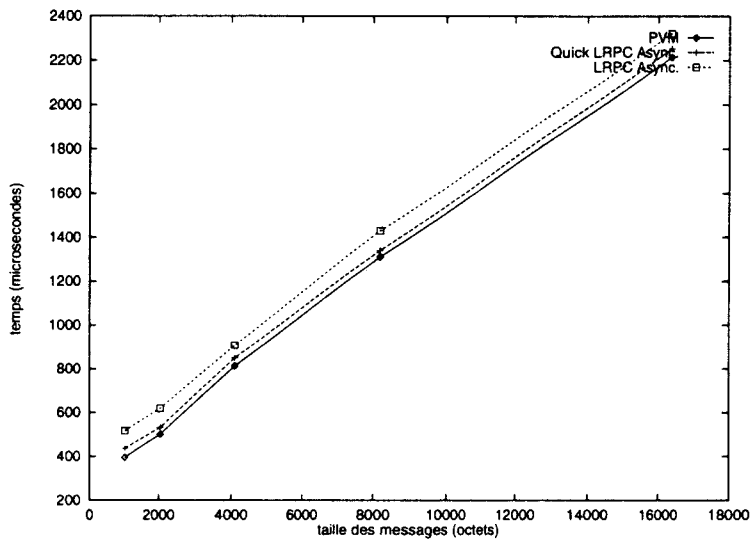

FIG. 6.12 - *Comparaison des performances des différentes variantes du mécanisme d'appel de procédure à distance asynchrone avec l'envoi de message PVM.*

Ces mesures montrent clairement que le surcoût occasionné par PM^2 par rapport à PVM est constant et ne dépend pas de la taille des données transmises, ce qui était prévisible puisque PM^2 n'effectue pas de recopie supplémentaire des données. L'écart entre les temps des LRPC avec ou sans création de processus est d'environ 80μs et comprend bien sûr le temps de création d'un nouveau processus mais aussi le temps nécessaire à l'initialisation de diverses structures de synchronisation et de mise en place de traitants d'exception dans la fonction intermédiaire exécutée par le processus. Enfin, l'écart entre le temps d'un envoi de message PVM et celui d'un « *Quick LRPC Asynchrone* » est d'environ 30μs, ce qui, rapporté au temps d'exécution d'un envoi de message de taille nulle, amène le **surcoût relatif maximum** de

PM² sur PVM à moins de 10 %, ce qui est très faible.

Pour évaluer les performances des appels synchrones, nous avons mesuré des programmes effectuant 1 000 itérations consistant à effectuer un appel synchrone (avec ou sans création de processus) dans un module distant. Le code associé au service déclenché était vide, la taille des arguments était identique à celle des résultats (indiquée en abscisse) et le processus appelant était déclaré « *non-migrable* ». Ces résultats (rapportés au temps d'un seul appel) sont comparés au temps d'exécution de deux envois de messages PVM de même taille (figure 6.13).

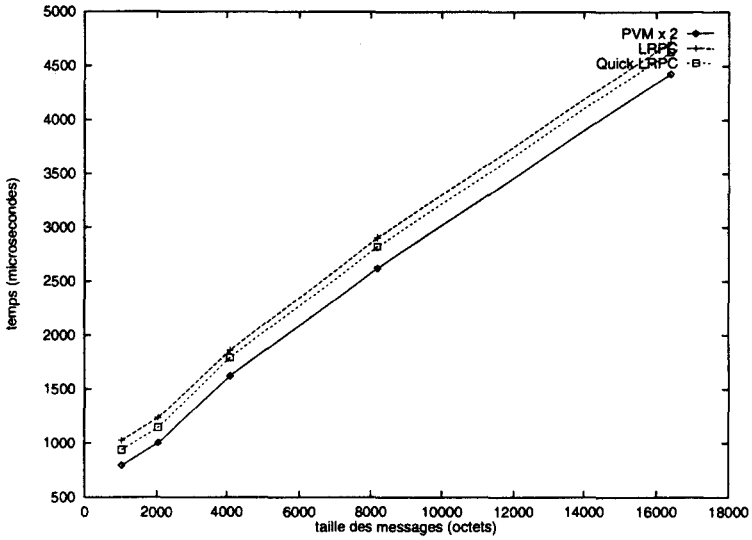

FIG. 6.13 - *Comparaison des performances des différentes variantes du mécanisme d'appel de procédure à distance synchrone avec un double envoi de message PVM.*

Là encore, les surcoûts observés sont indépendants de la taille des données transmises et l'écart entre les temps des LRPC avec ou sans création de processus est toujours d'environ 80μs. L'écart entre un « Quick LRPC » et un double envoi de message PVM est maintenant d'environ 150μs. Cet écart comprend notamment le temps passé dans la gestion de la structure d'attente établie sur le module du processus appelant.

6.4.4 Migration

Paradoxalement, le déroulement des opérations mises en jeu lors d'une migration de processus légers est beaucoup plus simple que dans le cas d'un appel de procédure à distance léger. Cette simplicité de mise en œuvre est due à la souplesse d'utilisation des fonctionnalités d'*hibernation de processus* fournies par la bibliothèque MARCEL. Une opération de migration de processus légers se déroule en trois étapes :

1. Avant de déclencher une migration de processus, il faut *geler* le module (`pm2_freeze`). Cette opération est directement déléguée à la bibliothèque MARCEL qui désactive[41] provisoirement le mécanisme de préemption automatique.

41. Cette désactivation est effectuée au moyen du simple positionnement d'une variable booléenne.

2. Lorsqu'un processus appelle la primitive `pm2_migrate_group`, un tampon de communication PVM est d'abord créé. Ensuite, pour chacun des processus « victimes », la primitive d'entrée en phase d'hibernation est appelée (`pthread_begin_hibernation_np`). Au cours de chacune de ses exécutions, cette primitive appelle une fonction définie dans PM^2 en lui passant les « blocs de glace » correspondant aux processus en hibernation. Dans cette fonction, chaque nouveau bloc de glace est empaqueté à la suite des précédents dans le tampon de communication courant.

   Éventuellement, des fonctions définies par l'application peuvent être appelées par PM^2 pour chacun des processus empaquetés, de façon à autoriser (par exemple) l'empaquetage de données annexes.

   Une fois les blocs de glace empaquetés, le message (de type `MIGRATION`) est émis vers le module cible, le module est *dégelé*.

3. Lorsque le message arrive sur le module cible, les blocs de glace sont désempaquetés chacun leur tour à des emplacements mémoire alloués par la bibliothèque MARCEL (pour bénéficier du mécanisme de cache de piles), puis les processus qu'il contiennent sont réveillés par la primitive `pthread_end_hibernation_np`.

   Là encore, des fonctions définies par l'utilisateur peuvent être appelées pour chacun des processus appelés (opérations post-migration).

6.4.4.1 Mesures

Pour évaluer les performances d'une opération de migration, nous avons mesuré le temps d'exécution d'un programme effectuant un *ping-pong* de processus léger. Pour éviter la prise en compte d'opérations de commutations de contexte parasites, le programme comportait un seul processus léger « applicatif » effectuant son auto-migration entre deux modules 1 000 fois consécutives (la mesure est ensuite rapportée à une seule migration). Les mesures ont été effectuées pour plusieurs **quantités de pile déplacées** différentes. Deux cas de figure ont été distingués : 1°) la migration d'un processus dont la taille de pile utilisée dépend uniquement de la taille de ses données locales (pas de récursivité) 2°) la migration d'un processus dont la taille de pile est directement proportionnelle à la profondeur d'appel de la fonction couramment exécutée (environ deux milles appels récursifs pour les migrations de 16 kilo-octets).

Dans le but d'observer un surcoût relatif maximal, les mesures ont été effectuées entre deux modules situés sur la même machine (Pentium120) et sont comparées aux temps d'envoi de messages PVM pour des données de même taille (figure 6.14).

Comme on peut le constater, une opération de migration appliquée à un processus comportant une profondeur d'appel quasi nulle introduit un surcoût constant par rapport à un envoi de message PVM, quelle que soit la taille des données impliquées dans la migration. Les choses sont différentes avec une taille de pile directement proportionnelle à la profondeur des appels et l'écart augmente avec la taille des données. Ce résultat n'est pas surprenant car il confirme celui mesuré dans la section traitant de l'extension de pile, indiquant que l'opération de translation de pile (utilisée dans la migration) possède une durée d'exécution proportionnelle à la profondeur des appels récursifs.

Notons pour terminer que la migration de processus demeure tout de même une opération très efficace et présente un coût relativement proche d'une opération de LRPC asynchrone (avec création).

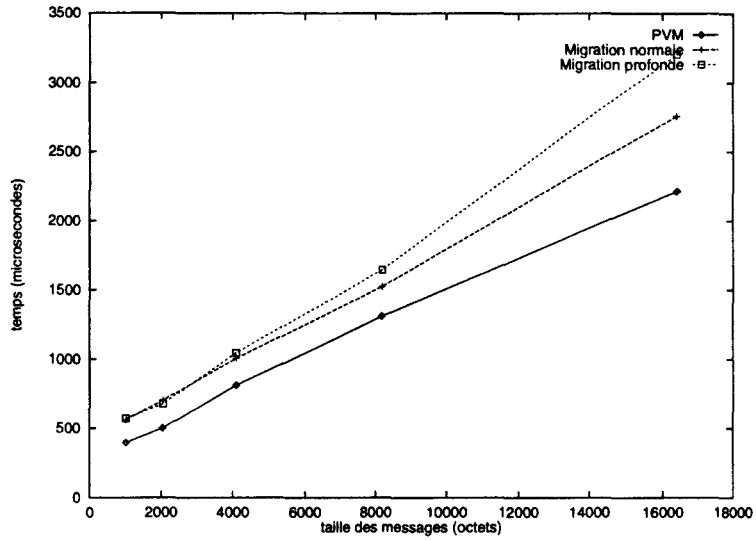

FIG. 6.14 - *Comparaison du temps d'exécution d'une opération de migration avec une opération d'envoi de message PVM déplaçant la même quantité de données.*

6.4.5 Clonage

L'implantation complète du mécanisme de *clonage léger* dans PM² n'est pas achevée au moment où nous écrivons ces lignes. Afin d'obtenir rapidement un prototype expérimental, nous en avons implanté une version simplifiée au niveau du protocole de *fusion* des résultats qui impose que le « survivant » de l'opération de fusion soit le processus ayant initié la séparation. Cette limitation nous a permis d'aboutir rapidement à une implantation opérationnelle nous permettant d'évaluer le gain en « expressivité du parallélisme » apporté par cette fonctionnalité. L'opération de clonage est donc actuellement implantée comme suit :

séparation Lorsqu'un processus exécute la primitive `SPLIT`, il commence par allouer (dans le tas) un tableau destiné à accueillir les résultats qui parviendront lors de l'opération de fusion. Ensuite, d'une manière similaire à une opération de migration, le processus envoie un « clone glacé » sur chacun des sites spécifiés. En fait, chacun de ces clones possède une caractéristique le distinguant des autres : son numéro de rang. Étant donné que PVM ne permet pas de modifier des données déjà empaquetées dans un tampon de communication, il faut donc renouveler l'opération d'empaquetage pour chacun des n clones, ce qui peut être lourd de conséquences sur les performances de cette opération. Dans l'implantation (restrictive) actuelle, le processus original est alors marqué *non-migrable*.

fusion Lorsqu'un clone exécute la primitive `MERGE`, il envoie simplement un message vers le site d'origine pour y déposer sa contribution et, dans le cas où il n'est pas le processus de rang 0, se suicide. Le processus original, quant à lui, se bloque sur un sémaphore en attendant l'arrivée de toutes les contributions, à l'issue de quoi il peut exploiter les résultats jusqu'à l'instruction `END_SPLIT`.

Nous avons évalué les performances « brutes » de ce mécanisme en mesurant le temps d'exécution total d'une opération de clonage (du `SPLIT` au `END_SPLIT`) pour un processus se

séparant en deux clones (chacun dans un module distinct). Les expériences ont été menées sur un Pentium120 (pour permettre des comparaisons avec les performances données dans les sections précédentes). Pour une taille de pile utile de 1 kilo-octet et un retour de résultat de 2 octets, le temps de l'opération de clonage est de 1044μs. En déduisant de ce nombre le temps d'une opération de migration (560μs dans les mêmes conditions) et le temps d'une opération d'envoi de message correspondant au retour de résultat (352μs), il reste un coût de 132μs qui résulte principalement de l'allocation (et de la libération) du tableau des résultats et de la synchronisation des différents processus.

On le voit, les performances exhibées par ce mécanisme de clonage « à sémantique réduite » sont très encourageantes, car elles sont à peine moins bonnes que celles d'un appel de procédure à distance léger synchrone. Il reste maintenant à terminer la réalisation de ce mécanisme en implantant la phase de *fusion* telle que décrite au chapitre 5. De ce point de vue, nous optons actuellement pour une « centralisation » des résultats partiels sur le site du processus original, qui permettra de minimiser le nombre de messages échangés pour déterminer le processus survivant et lui acheminer ces résultats.

6.5 Conclusion sur la réalisation

Dans ce chapitre, nous avons décrit l'implantation actuelle de l'environnement PM^2 ainsi que celle d'une bibliothèque sur laquelle l'environnement s'appuie: la bibliothèque de processus légers MARCEL. Nous avons également brièvement présenté la bibliothèque PVM que l'environnement utilise non seulement pour la réalisation des communications entre les modules, mais également pour la gestion (dynamique) des configurations.

Pour chacune des fonctionnalités principales de ces trois composantes, nous avons effectué des mesures de performances. En ce qui concerne les communications, nous avons évalué le « surcoût » introduit par l'environnement PM^2 par rapport à la bibliothèque native (*i.e.* PVM) et montré qu'il était faible et indépendant du volume d'informations transportées entre les modules. En ce qui concerne la gestion des processus légers, nous avons montré la remarquable efficacité de la bibliothèque MARCEL due en partie à sa vocation dédiée au calcul intensif.[42]

Si l'on considère le compromis **fonctionnalités/portabilité/efficacité** mentionné à la fin du chapitre 4, nous pensons que l'environnement PM^2 occupe une place intéressante dans le « cercle » des environnements à base de processus légers car il ne néglige aucune des caractéristiques suivantes :

fonctionnalités Au sens large, PM^2 ne fournit pas un nombre de fonctionnalités astronomique, en comparaison avec des environnements tels que Chant ou TPVM. Cependant, si l'on raisonne en terme de fonctionnalités **de base**, l'environnement PM^2 figure parmi les tous premiers de sa catégorie en fournissant des garanties d'ordonnancement au sein de chaque module (préemption, priorités, commutations efficaces), des mécanismes de migration et de clonage de processus entre machines homogènes, des mécanismes d'extensions dynamiques des piles d'exécution et des mécanismes (encore élémentaires) de sauvegarde de processus sur disque.[43]

portabilité Avec une réalisation s'appuyant sur la bibliothèque de communication PVM, la portabilité de l'environnement PM^2 sur de nouvelles architectures repose principalement

42. Et à son implantation au niveau utilisateur.

43. En plus des mécanismes traditionnels de gestion des processus et des communications, évidemment.

sur celle de la bibliothèque MARCEL. Nous avons vu, dans la section qui lui est consacrée, que cette bibliothèque a déjà été portée sur six architectures et que le portage sur un nouveau système (vraisemblablement la machine CAPITAN du LIP) ne nécessite l'adaptation que d'une très faible portion du code source. En fait, la réalisation des fonctionnalités mentionnées au point précédent n'a pas été obtenue au détriment d'une trop forte réduction de la portabilité, puisque la seule concession faite dans ce domaine concerne la dépendance vis-à-vis du compilateur C de la fondation GNU.

efficacité En ce qui concerne la gestion des communications, la réalisation de PM^2 adopte typiquement un compromis (concrétisé par le choix de PVM) favorisant les *fonctionnalités* et la *portabilité* sur l'*efficacité*. Sans s'avérer très pénalisante en pratique (comme nous le verrons au chapitre suivant), l'efficacité de la bibliothèque PVM (du moins en ce qui concerne la version du domaine public) souffre d'une réalisation occasionnant de nombreuses recopies de données et se voit aujourd'hui, dans ce domaine, assez nettement dépassée par MPI, le standard actuel en matière d'interface aux communications par envoi de message.

En revanche, la gestion des processus légers dans l'environnement PM^2 a été étudiée de façon à représenter, dans ce cadre précis d'utilisation, une solution sans compromis. En effet, nous avons vu que la bibliothèque MARCEL présentait à la fois des fonctionnalités évoluées, un degré de portabilité élevé et des performances remarquables. Cette caractéristique provient principalement d'une phase de conception s'inscrivant complètement dans le cadre du projet ESPACE ayant permis une évaluation précise des fonctionnalités nécessaires et du même coup une réalisation s'affranchissant des contraintes posées par une hypothétique vocation « multi-usages » de la bibliothèque.

L'implantation de l'environnement PM^2 est maintenant relativement stabilisée[44] et est utilisée dans plusieurs équipes de recherches françaises pour le développement d'applications parallèles distribuées. Mis à part le fait d'être opérationnelle et donc de permettre à de nombreuses expérimentations (notamment en matière d'équilibrage dynamique de charge) d'être menées, cette implantation a pour principal mérite de répondre positivement aux exigences du modèle de programmation PM^2 :

Virtualisation En permettant la création de plusieurs milliers de processus légers par nœud et en assurant leur gestion à faible coût, l'environnement PM^2 permet d'effectuer un pas important vers la notion de *machine virtuelle*. À cet égard, les résultats d'expérimentations, montrant que l'augmentation du nombre de processus légers au sein d'une application[45] ne dégrade pas ses performances, sont très importants car ils représentent une première validation de notre approche.

Concurrence En instaurant, au sein de chacun des modules d'une application, un ordonnancement préemptif avec gestion des priorités dont l'efficacité est indépendante du nombre de processus légers, l'environnement PM^2 effectue un pas supplémentaire vers cette notion de machine virtuelle. Les expérimentations effectuées mettent en évidence le très faible coût lié à la « préemptivité » de l'ordonnancement et confirment son adéquation au support d'un degré de parallélisme important.

44. Sauf en ce qui concerne le mécanisme de clonage léger, qui constitue une extension récente.
45. Au sein d'un seul module

Mobilité En fournissant un mécanisme efficace de migration de processus[46] venant épauler le mécanisme d'appel de procédure à distance léger pour l'exploitation des architectures distribuées, l'environnement PM^2 fournit des outils permettant d'étendre la *virtualisation* offerte au niveau de chaque nœud (ou module) à une virtualisation concernant l'ensemble de l'architecture distribuée. La réalisation de ces mécanismes montre que les problèmes techniques sont maîtrisés et que leur mise en œuvre peut être effectuée dans une librairie de niveau utilisateur de manière efficace.

Ce chapitre nous a donc permis de montrer la « faisabilité » du modèle de programmation PM^2 ainsi que les bonnes performances de son implantation. Dans le chapitre suivant, nous examinerons plusieurs applications développées avec PM^2 et constaterons l'adéquation des fonctionnalités offertes par PM^2 pour leur support efficace.

46. Entre les modules d'une configuration homogène.

Chapitre 7

PM^2 : quelques applications

Les deux chapitres précédents ont introduit bon nombre d'arguments justifiant l'intérêt du modèle de programmation PM^2 pour le support des applications irrégulières et ont montré les bonnes performances « brutes » de l'implantation actuelle. Ce chapitre apporte une confirmation de ces caractéristiques, en examinant quelques applications parallèles développées avec l'environnement PM^2 et en montrant les avantages issus de cette approche, tant sur le plan de la conception que sur le plan de l'efficacité.

Un point important à souligner est que ces applications ont été conçues et développées soit par des équipes de recherche « extérieures » au projet ESPACE, soit en collaboration étroite avec de telles équipes. Ces collaborations ont notablement contribué aux évolutions de l'environnement PM^2, ce qui était un objectif du projet ESPACE. Parmi les différentes applications développées avec PM^2, nous avons choisi de n'en présenter que trois[1] :

- Les applications d'optimisation combinatoire ont eu un impact important sur des choix de conception de PM^2 (priorités, virtualisation). Une implantation d'un des grands classiques de l'optimisation combinatoire, le problème du voyageur de commerce, est présenté dans la première section. Cette application a été développée par Yves Denneulin (LIFL) et les premiers résultats obtenus s'avèrent très encourageants.

- L'environnement PM^2 peut également se révéler être un excellent candidat pour le support d'applications à parallélisme de données. Les équipes de Lyon (C. Perez, L. Bougé) et Lille (B. Kokoszko, J. Soula, J.L. Dekeyser) étudient des modèles d'exécution « multithread » pour les programmes data-parallèles. En particulier, Christian Perez (LIP) a très longuement étudié et évalué les avantages de l'utilisation des threads pour implanter les processus virtuels HPF. Son étude et ses principaux résultats sont présentés dans la seconde section.

- La plate-forme PM^2 a été particulièrement conçue pour offrir des fonctionnalités aidant à la conception de régulateurs de charge. Yves Denneulin utilise les fonctionnalités de migration et de gestion des priorités pour définir le régulateur LBMP (*Load-Balancing with Migration directed by Priorities*). Nordine Melab (LIFL), a également travaillé avec PM^2 pour définir un outil de régulation de charge adaptatif dans lequel la politique d'information s'adapte à la charge courante de la machine. Nous présentons ce travail dans la troisième section.

1. Nous reviendrons sur les autres en conclusion de ce chapitre.

7.1 Optimisation combinatoire

À l'occasion de notre participation à l'opération inter-PRC STRATAGÈME [149], une collaboration suivie s'est établie avec l'équipe de recherche PNN dirigée par Catherine Roucairol (PR*i*SM, Versailles). Cette collaboration nous a permis de mieux appréhender le domaine de l'optimisation combinatoire et les problèmes posés par la parallélisation des applications qu'il inclut, puisque la décomposition parallèle de ces dernières s'avère *totalement imprévisible*.

Plus précisément, Yves Denneulin (équipe GOAL, Lille), dont le travail de thèse consiste à concevoir et à réaliser un équilibreur de charge pour ces applications[2] [54], a mené un certain nombre d'expérimentations qui ont influencé la conception de l'environnement PM^2. Par exemple, ces expérimentations nous ont fait comprendre l'intérêt de pouvoir attacher des priorités aux différentes tâches (*i.e.* processus légers) de manière à « guider » le déroulement global de l'exécution en fonction d'heuristiques définies par le programmeur. Nous reviendrons sur ce point en section 7.1.1.3.

Dans la suite de cette section, nous présentons un exemple bien connu d'application d'optimisation combinatoire (le voyageur de commerce) développée au-dessus de l'environnement PM^2 par Yves Denneulin.

7.1.1 Problème du voyageur de commerce

L'application présentée ici est une version parallèle d'un algorithme résolvant le problème du voyageur de commerce (problème NP-complet). Le principe de l'algorithme séquentiel est assez simple et consiste en l'exploration récursive (en profondeur d'abord) d'un espace de recherche. À chaque étage de l'arbre d'exploration résultant, notre voyageur de commerce peut prendre la direction d'un certain nombre de villes non encore visitées, mais pas « toutes » les villes non-visitées car il mémorise le plus court chemin qu'il a déjà réussi à dénicher jusqu'alors et rebrousse chemin dès qu'il ne peut pas améliorer son itinéraire (élagage de l'arbre). L'algorithme s'arrête lorsque toutes les pistes ont été explorées.

La parallélisation efficace d'un tel algorithme pour une architecture distribuée est difficile, comme il a été évoqué au chapitre 2 (section 2.2.5.2). En particulier, la gestion des « nœuds » de l'arbre en attente d'être explorés constitue souvent un point épineux dans les implantations classiques de ce type d'application.

7.1.1.1 Principe de l'implantation actuelle

Conformément au modèle de programmation prôné par les concepteurs de PM^2, l'implantation parallèle de cet algorithme n'utilise aucune structure de donnée globale (file de priorité ou autre) pour le stockage des nœuds en attente d'exploration. En effet, les nœuds, qui constituent les tâches parallèles de l'application, ne sont **jamais** mis en attente. Au contraire, leur exécution démarre (*i.e.* un nouveau processus léger est créé) dès qu'ils sont générés.[3]

Dans l'état actuel de l'implantation (la réalisation du régulateur dynamique de charge LBMP[4] n'étant pas achevée), la stratégie de régulation adoptée est une politique de répartition cyclique des processus légers sur les nœuds. Par conséquent, les résultats présentés dans la section suivante s'entendent « sans régulation dynamique ».

2. Au-dessus de l'environnement PM^2.

3. Enfin presque, car il faut tout de même décompter le temps nécessaire à leur acheminement sur le nœud choisi pour supporter leur exécution.

4. Load Balancing with Migration directed by Priorities

En réalité, chaque nœud de l'arbre n'est pas pris en charge par un nouveau processus léger, car la granularité du traitement associé à un nœud est vraiment infime. Un contrôle de la granularité est effectué au sein du programme pour assurer que chaque processus léger soit responsable de l'exploration d'un sous-arbre (dont la taille varie au cours de l'application) de l'arbre global [53]. Lorsqu'un processus léger est sur le point de « sortir » de son domaine d'exploration, alors de nouveaux processus légers sont créés (par appels de procédure à distance).

L'élagage de l'arbre s'effectue de la même manière que dans la version séquentielle, à savoir grâce à la mémorisation du plus court itinéraire rencontré jusqu'alors. Cette valeur est conservée dans une variable globale répliquée dans tous les modules, et mise à jour de façon non-atomique.[5] Lorsqu'un processus léger détecte un itinéraire plus court que ceux rencontrés jusqu'alors (d'après sa connaissance), il met à jour toutes les répliques de cette variable à l'aide d'une série d'appels de procédure à distance asynchrones.

La section suivante présente les performances de cette implantation sur des instances de problème de taille 15 et 16.

7.1.1.2 Mesures

La figure 7.1 montre les temps d'exécution de cette application pour trois instances de problème différentes (un de taille 15 et deux de taille 16) sur la ferme de processeurs ALPHA de l'université de Lille. Les mesures ont été effectuées sur des configurations variant de 1 à 15 processeurs.

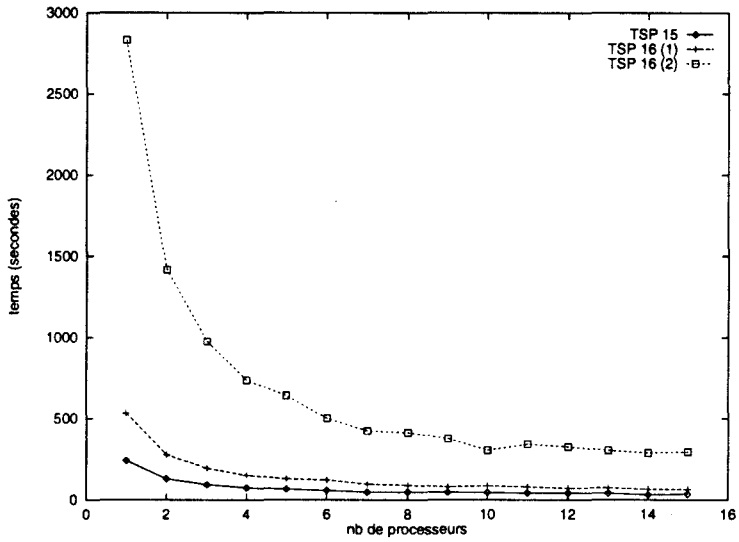

FIG. 7.1 - *Performances de trois instances du TSP (Travelling Salesman Problem) sur un nombre de processeurs variant de* 1 *à* 15.

Sur cette figure, il est possible d'observer un facteur d'accélération de 6,7 sur 15 processeurs pour l'instance de taille 15, de 8,3 pour le premier problème de taille 16 et de 9,6 pour le second. Sans être catastrophiques, ces accélérations peuvent cependant être qualifiées de

5. Le maintien d'une cohérence forte de la valeur de cette variable serait vraisemblablement plus coûteux que le gain obtenu en contrepartie.

« moyennes ». Pour bien les interpréter, il est nécessaire d'examiner le nombre de processus légers impliqués dans chacune de ces expériences.

En fait, la granularité des sous-arbres a volontairement été réduite ici, de manière à étudier le comportement « extrême » de PM^2 dans une telle situation. Ainsi, l'exécution de la seconde instance du problème de taille 16, qui dure 295 secondes sur une configuration de 15 processeurs, a vu la création de **plus d'un million de processus légers** dans cet intervalle de temps, ce qui correspond à une espérance de vie de moins de 4 millisecondes par processus léger. Dans ces conditions, les résultats exhibés par cette expérience deviennent très bons, notamment en regard de la simplicité de développement de l'application et de l'absence de mécanisme d'équilibrage dynamique de charge.

7.1.1.3 Perspectives

On le voit, malgré les conditions difficiles, l'implantation du TSP (*Travelling Salesman Problem*) parallèle décrite dans cette section exhibe des performances honorables pour une facilité d'expression réellement remarquable, ce qui semble indiquer la bonne adéquation de l'environnement PM^2 au support des applications fortement irrégulières.

Des travaux concernant la conception d'un ordonnanceur global de processus légers dans le cadre des applications d'optimisation combinatoire sont actuellement menés par Yves Denneulin : il s'agit de l'ordonnanceur LBMP (*Load Balancing with Migration directed by Priorities* [54]) dont l'objectif est d'assurer une répartition des processus légers qui fasse en sorte que leur vitesse de progression soit conforme à la priorité qui leur est attachée.

La notion de priorité d'exécution est en effet très utile dans le cadre des applications « guidées par heuristiques » (comme c'est le cas ici) car elle permet de favoriser par exemple l'exploration de certains sous-arbres sans nécessiter le recours à des mécanismes de files de priorité. Bertrand Le Cun (équipe PNN, Versailles), qui a effectué un portage de la bibliothèque BOB [46] au-dessus de l'environnement PM^2, a observé le très bon comportement d'une exploration effectuée par plusieurs processus légers concurrents lorsque des priorités sont attachées à ceux-ci (expérience menée en contexte monoprocesseur). En particulier, le nombre de nœuds explorés était inférieur à celui obtenu par un parcours « meilleur d'abord » purement séquentiel.

C'est cette propriété que nous voulons exploiter au travers de l'outil LBMP, en l'étendant aux configurations distribuées. Ce travail sera décrit dans la thèse d'Yves Denneulin.

7.2 Support pour le parallélisme de données

Comme nous avons pu le constater dans les points précédents, les premières applications irrégulières développées au-dessus de l'environnement PM^2 sont essentiellement caractérisées par un *parallélisme de contrôle* très important et un volume de données manipulées assez faible. Aussi était-il intéressant d'élargir le domaine d'utilisation de PM^2 aux applications à *parallélisme de données* qui nécessitent une démarche de conception sensiblement différente.

Dans ce domaine, nous avons eu l'opportunité d'établir des collaborations avec deux équipes de recherche françaises : l'équipe PARADIGME du Laboratoire de l'Informatique du Parallélisme de Lyon dirigée par Luc Bougé et l'équipe WEST du Laboratoire d'Informatique Fondamentale de Lille dirigée par Jean-Luc Dekeyser.

Dans les sections suivantes, nous présentons les travaux de l'équipe de Lyon concernant l'intégration de mécanismes de régulation dynamique de charge dans des programmes HPF.

7.2.1 HPF et régulation de charge

Le langage HPF (*High Performance Fortran* [94]) est un langage à parallélisme de données basé sur le langage Fortran. Sa principale particularité est d'inclure de nombreuses possibilités d'optimisation du code généré, notamment au moyen de directives spécifiées par le programmeur. Par exemple, il est possible de contrôler la répartition des données en exprimant des contraintes d'*alignement* de tableaux dans des modèles (*templates*), de *distribution* de modèles sur des processeurs virtuels et de *placement* de ces processeurs virtuels sur des processeurs physiques.

Si l'efficacité du code HPF n'est plus à démontrer dans un cadre applicatif strictement régulier, il n'en est évidemment pas de même si les traitements appliqués aux données sont de nature irrégulière. Dans ce cas, des mécanismes permettant de réguler la charge sur les processeurs sont nécessaires. C'est la raison pour laquelle HPF v2.0 [95] intègre par exemple des possibilités de distribution « irrégulière » des modèles sur les processeurs virtuels. Cependant, cette approche demeure encore trop restrictive et reste principalement à la charge du programmeur. Dans ce cadre, l'introduction de mécanismes de régulation dynamique de charge dans les applications HPF revêt un intérêt particulier.

C'est précisément le thème de l'étude menée par Christian Perez (LIP) dans le cadre de sa thèse [139, 138]. L'idée centrale de cette étude est d'utiliser les processus légers distribués pour supporter la notion de *processeur virtuel HPF* définie par le langage. Pour cela, l'environnement PM^2 a été choisi, principalement parce qu'il fournit, via le mécanisme de *migration de processus léger*, un outil de régulation de charge à la fois souple et efficace.

Nous allons maintenant présenter succinctement la démarche de Christian Perez.[6] Tout d'abord, nous rappelons la notion générale de *processeur virtuel* dans le langage HPF, à la suite de quoi nous examinons pourquoi les processus légers sont de bons candidats pour supporter la notion de processeur virtuel. Dans un second temps, nous évoquerons l'étude préliminaire concernant les performances de l'environnement PM^2 puis décrirons les expérimentations menées sur une application test. Enfin, nous rapportons les conclusions de ces expérimentations qui montrent l'intérêt des processus légers et de l'environnement PM^2 pour le support des applications irrégulières.

7.2.1.1 HPF et processeurs virtuels

Dans le langage HPF, la répartition des données s'effectue à l'aide de trois niveaux d'abstraction (figure 7.2) :

Alignement Les tableaux d'une application peuvent être « *alignés* » entre eux par le biais d'un *modèle* (grille multidimensionnelle). Cette opération assure que les éléments de tableaux alignés aux mêmes emplacements dans la grille se trouveront sur le même processeur physique au bout du compte.

Distribution Les modèles peuvent, à leur tour, être *distribués* sur un ensemble de processeurs virtuels (grille multidimensionnelle). Typiquement, les politiques de distribution disponibles sont la distribution « *par bloc* » et la distribution « *cyclique* ». Notons que la version 2.0 de HPF fournit des distributions plus souples, telles qu'une distribution par bloc généralisée (autorisant une taille de bloc variable), pour mieux appréhender les programmes irréguliers.

6. Le lecteur intéressé pourra se reporter aux documents [139] et [138] pour de plus amples informations.

Placement La dernière étape de cette répartition des données consiste à établir la correspondance entre les processeurs virtuels du langage et les processeurs physiques de l'architecture, en particulier lorsque le nombre de processeurs physiques est inférieur au nombre de processeurs virtuels.

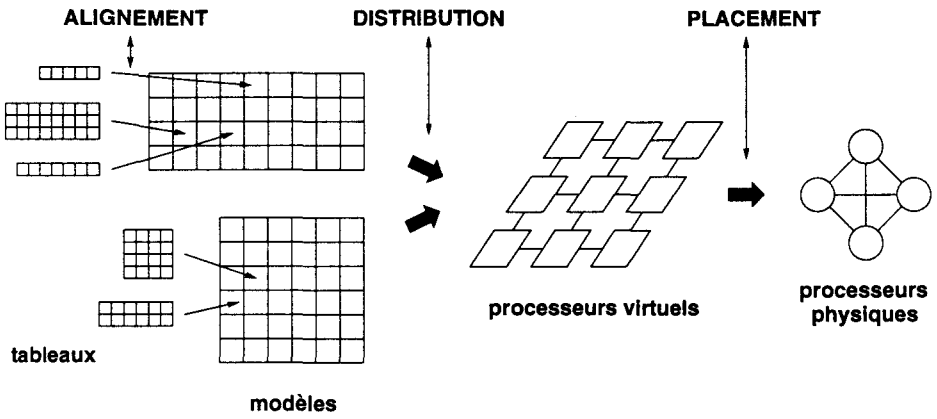

FIG. 7.2 - *Modèle de répartition des données dans le langage HPF.*

Les différents travaux concernant l'extension de HPF pour le support des applications irrégulières proposent principalement un enrichissement de la phase de distribution des données (qui peuvent d'ailleurs être redistribuées dynamiquement) de manière à contrôler finement la répartition des données. Cependant, ce travail reste à la charge du programmeur et complique fortement la phase de développement des applications.

La proposition faite par l'équipe de Lyon est d'étendre la phase de *placement*[7] des processeurs virtuels en autorisant leur répartition dynamique au cours de l'exécution d'un programme. Le programmeur peut alors guider cette répartition s'il connaît le comportement de l'application ou laisser le système décider de la stratégie de régulation à mettre en œuvre en tenant compte de facteurs externes tels que la charge des processeurs ou même leur puissance.

La notion de *processeur virtuel* étant l'unité de base pour la régulation de charge, le point clé de cette approche consiste à définir une implantation réalisant un bon compromis entre facilité de mise en œuvre, efficacité et occupation mémoire.

7.2.1.2 Virtualisation avec les processus légers

Trois approches différentes ont été évaluées en ce qui concerne l'implantation des processeurs virtuels : 1°) la prise en charge par un processus lourd, 2°) l'émulation par une boucle dite « de virtualisation » et 3°) la prise en charge par un processus léger.

La prise en charge d'un processeur virtuel par un processus lourd présente l'avantage d'être triviale à mettre en œuvre[8]. En revanche, elle pêche sur le plan de l'efficacité principalement à cause de la lourdeur des communications locales à un processeur physique et à cause de l'opération de migration d'un processeur virtuel qui, par le biais d'un environnement tel que MPVM [25] ou Cocheck [152], peut durer plusieurs secondes (cf. section 5.4.6).

7. Qui devient alors *dynamique*.

8. Les compilateurs HPF actuels peuvent générer autant de processus que de processeurs virtuels.

L'émulation des processeurs virtuels par des boucles de virtualisation constitue, quant à elle, l'approche optimale en ce qui concerne les opérations de migration d'un processeur physique à un autre (seules les données sont transférées), mais présente en revanche l'inconvénient d'être très délicate à mettre en œuvre, notamment parce qu'il faut mettre en place des mécanismes de synchronisation pour respecter certaines règles de dépendance. En fait, cette approche revient à simuler l'exécution concurrente de multiples flots d'exécution au sein d'un processus lourd, c'est-à-dire à implanter le fonctionnent d'un mini-ordonnanceur de processus légers au beau milieu du code de l'application.

La troisième approche, qui consiste à utiliser des processus légers (un par processeur virtuel) pour assurer la virtualisation définie au niveau du langage, semble représenter un très bon compromis entre les deux approches précédentes. Pour le vérifier, des études préliminaires ont été menées. Nous les résumons dans la section suivante.

7.2.1.3 Étude préliminaire

Afin d'évaluer la pertinence d'une introduction des processus légers dans un code HPF compilé, autrement dit pour vérifier que le surcoût dû à leur gestion ne pénalise pas l'exécution d'un programme régulier, Christian Perez a effectué un certain nombre de tests [139] destinés à mesurer les coûts opératoires d'un certain nombre de fonctionnalités. Nous ne citons ici que les plus significatives.

Tout d'abord, la « perturbation » occasionnée par l'environnement PM^2 sur un programme séquentiel a été mesurée, et ce pour différentes périodes de préemption. Les résultats obtenus sont parfaitement conformes aux nôtres (donnés en section 6.4.2) et confirment que le surcoût introduit par la scrutation périodique du *processeur serveur* est de l'ordre de 2 % sur une station Sparc5 avec une préemption de 20 millisecondes.

Ensuite, le coût général de la multiprogrammation a été évalué, en faisant varier le nombre de processus légers coopérant à la résolution d'un travail fixé (en l'occurrence des calculs faisant intervenir des tableaux). Là encore, les résultats sont en accord avec ceux que nous avons fournis dans la section consacrée à l'implantation du noyau MARCEL : 1°) le nombre de processus légers utilisés influence très peu le temps d'exécution de l'application (moins de 1 % sur une Sparc5 avec 64 processus légers) et 2°) le temps d'exécution diminue parfois alors que le nombre de processus augmente, ce qui est dû à des effets de cache favorisés par la « localité » des données manipulées par chaque processus léger.

Enfin, l'efficacité du mécanisme de migration de processus légers a été testée et s'avère être en parfait accord avec nos mesures. Christian Perez observe d'ailleurs un phénomène *a priori* surprenant, en mesurant un débit supérieur pour l'opération de migration que pour celle d'appel de procédure à distance. En fait, dans le cadre précis de l'expérience, ce phénomène est principalement dû aux primitives d'empaquetage utilisées par chacun de ces deux mécanismes (`pvm_pkbyte` contre `pvm_pkint`).

La conclusion de cette pré-étude était donc très favorable à l'utilisation des processus légers pour l'encapsulation des processeurs virtuels de HPF, car le coût occasionné par leur utilisation s'est avéré très faible. La section suivante montre les premiers résultats obtenus par une telle utilisation.

7.2.1.4 Expérimentation sur une application : élimination de Gauss

L'objectif étant, pour le moment, d'évaluer la faisabilité de l'approche, ces travaux n'ont pas été intégrés dans un compilateur et portent de ce fait sur des programmes « étendus manuellement » faisant appel à une bibliothèque de fonctionnalités maison. Plus précisément, le compilateur Adaptor (HPF vers Fortran77 [15]) est utilisé pour obtenir un code Fortran parallélisé, puis quelques modifications (intégration des processus légers) sont effectuées « à la main » et enfin le compilateur Fortran77 est utilisé pour obtenir les programmes exécutables (*multithreadés*).

En raison de la nécessité de pouvoir retrouver, à partir d'un processus léger, les tableaux qu'il gère, l'organisation mémoire des processus lourds est assez particulière et contient des tableaux spéciaux contenant, pour chaque processus léger, l'adresse de base des tableaux qu'il manipule. Ces informations sont mises à jour lors de la création des processus légers, ainsi que lors de leur migration. À ce propos, il est intéressant de noter qu'il est fait intensivement usage des primitives pré- et post-migration de l'environnement PM^2 de manière à assurer le déplacement des tableaux (alloués dans le tas) en cas de migration du processeur virtuel qui les contient.

L'application choisie pour effectuer les premières évaluations est un programme de résolution de systèmes d'équations par élimination de Gauss avec pivot partiel. Ce problème est un problème semi-régulier dans le cas d'une répartition par bloc des colonnes des matrices, mais devient régulier avec une répartition cyclique. Pour mesurer l'efficacité que peut avoir un régulateur dynamique de processeurs virtuels, une répartition par bloc a donc été choisie, sachant que la répartition cyclique constituera le « temps de référence » en quelque sorte.

La stratégie de migration utilisée (au moyen de migrations de processus) dans la version « régulée » est une stratégie ad-hoc, utilisant une heuristique liée au rang du pivot. La figure 7.3 indique les temps d'exécution obtenus pour la version du programme avec et sans équilibrage de charge. Ces temps sont comparés avec les temps d'exécution des programmes obtenus avec le compilateur Adaptor (distribution cyclique et par bloc).

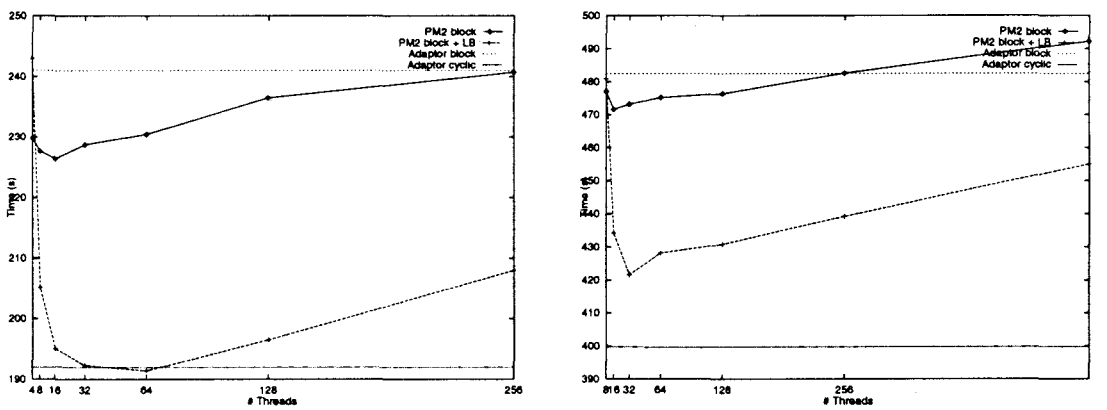

FIG. 7.3 - *Temps d'exécution du programme « élimination de Gauss » pour une matrice* 2048 × 2048 *sur* 4 *processeurs ainsi que pour une matrice* 3072 × 3072 *sur* 8 *processeurs.*

Ces mesures mettent en évidence un gain atteignant 17 % apporté par la régulation dynamique de charge par rapport à une version du programme non-régulée, ce qui est un bon résultat. De plus, par comparaison avec le temps de référence établi par la version « Adaptor

cyclique », on ne décèle qu'un écart s'élevant jusqu'à 7 %, ce qui est également un bon résultat. À ce propos, les auteurs rapportent qu'il est surtout dû à la lenteur des communications qui peuvent rendre insignifiants les petits déséquilibres de calculs.

7.2.1.5 Conclusion

Le travail que nous avons décrit ici propose l'introduction d'un mécanisme de régulation dynamique de charge dans le langage HPF basé sur l'utilisation des *processeurs virtuels* en tant qu'unité de migration entre les processeurs physiques. La faisabilité de l'approche a été montrée en implantant les processeurs virtuels à l'aide de processus légers. L'évaluation de cette implantation sur une application d'*élimination de Gauss avec pivot partiel* a exhibé des performances assez prometteuses.

7.3 Régulation de charge adaptative

Nous avons évoqué, dans les chapitres précédents, la nécessité des mécanismes d'*équilibrage de charge dynamique* pour l'exécution efficace des applications fortement irrégulières. La section précédente a d'ailleurs présenté une approche représentative dans ce domaine, qui utilise le mécanisme de migration de processus légers comme outil central pour l'équilibrage de charge des applications.

Dans cette section, nous présentons une approche adaptative pour la régulation de charge dans les applications irrégulières. Ce travail, mené au sein de l'équipe de recherche PALOMA (Lille) dirigée par Bernard Toursel, constitue le travail de thèse de Nordine Melab [117] (projet PARALF).

7.3.1 Le régulateur ALBA

ALBA (*Adaptative Load Balancing Algorithm*) est un algorithme adaptatif de régulation de charge pour architectures distribuées [119, 118]. L'objectif de cet algorithme est de fournir à chacun des modules d'une application une vision (plus ou moins précise) de la *charge globale* de la machine, de façon à ce que chaque module puisse, par exemple, choisir les modules les moins chargés lors d'une phase de distribution de travail. Dans une application « ALBA », toutes les décisions de transfert et de répartition de travail peuvent donc être prises de manière décentralisée.

Pour ce faire, l'environnement comprend un *agent d'information central* chargé de la récolte et de la diffusion des informations de charge dans une application. Plus précisément, la récolte des informations de charge est une opération passive du point de vue de cet agent, car ce sont des *agents d'informations* décentralisés (un par module) qui prennent l'initiative de communiquer leur charge à l'agent central (figure 7.4).

L'originalité de cette approche réside dans l'utilisation de *fréquences* de communication des informations de charge *adaptatives*. En effet, tant au niveau de l'agent central qu'au niveau des agents locaux, les *délais* écoulés entre chaque « envoi de message » sont réajustés à chaque étape afin d'adapter la fréquence de mise à jour des informations à l'état de charge global du système. L'objectif de ce mécanisme étant d'introduire une perturbation minimale dans les applications, la progression mathématique de ces fréquences est inversement proportionnelle à un coefficient de charge moyenne du système calculé par l'agent central.[9]

9. Le lecteur intéressé pourra se reporter à [118] pour de plus amples informations

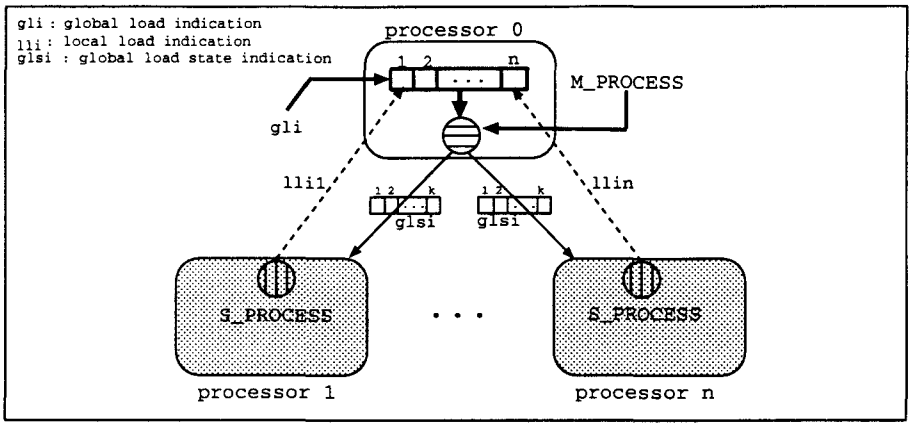

FIG. 7.4 - *Structure de l'agent d'information central du système ALBA.*

La réalisation actuelle du système ALBA [118] s'appuie sur l'environnement PM², ce qui permet d'implanter les agents locaux par des processus légers s'exécutant concurremment avec les processus applicatifs au sein des différents modules. L'agent central, quant à lui, est implanté par un module PM². Les transmissions d'informations de charge s'effectuent, dans un sens comme dans l'autre, en utilisant des appels de procédure à distance légers *asynchrones*.

7.3.1.1 Expérimentation sur le problème du taquin

Une évaluation du système ALBA a été effectuée sur une implantation parallèle d'une application irrégulière bien connue : la résolution du « jeu de taquin » (figure 7.5) par l'algorithme IDA* [104].

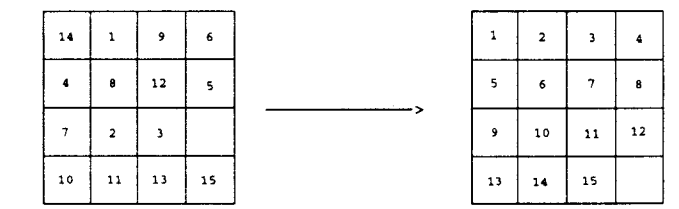

FIG. 7.5 - *Le problème du taquin consiste à déterminer la suite de déplacements élémentaires permettant d'aboutir à la configuration finale à partir d'une configuration initiale donnée.*

Cette implantation, évidemment développée au-dessus de PM², est basée sur la création d'une arborescence de processus légers explorant l'espace de recherche des solutions. Comme dans l'application du voyageur de commerce examinée au début de ce chapitre, chaque processus léger explore un sous-arbre de l'espace de recherche, puis génère éventuellement de nouveaux processus lorsqu'un seuil de granularité fixé est atteint (taille maximale d'un sous-arbre). Cette génération de nouveaux processus légers s'effectue au moyen du mécanisme de LRPC à attente différée et utilise les informations de charge délivrées par le système ALBA pour le choix des sites destinataires.

La section suivante relate les premiers résultats obtenus à l'aide de cette stratégie.

7.3.1.2 Mesures

L'évaluation préliminaire du système ALBA a été effectuée sur l'application du jeu de taquin détaillée précédemment. Le tableau 7.1 présente les temps d'exécution de cette application sur la ferme de processeurs ALPHA du LIFL (16 processeurs) avec différentes instances de problèmes de taille 15.

TAB. 7.1 - *IDA* appliqué au problème du taquin de taille 15 : résultats préliminaires*

classe	**configurations et temps d'exécution séquentielle**	**accélération relative moyenne**	**efficacité**
petit	28(11.3s), 78(20.5s), 23(29.7s)	6.45	0.40
normal	27(9.2 min), 32(20.5 min), 63(29.6 min)	12.45	0.77
gros	60(1h45), 82(2h52), 88(3h19)	14.83	0.92

Ces mesures montrent des facteurs d'accélération intéressants, notamment pour les problèmes les plus gros. Si elles montrent bien l'efficacité de cette implantation parallèle, elles ne mettent pas vraiment en évidence, en revanche, l'intérêt de l'aspect *adaptatif* de la régulation qui nécessite encore d'être comparé avec d'autres stratégies de régulation (par exemple cyclique ou aléatoire).

7.3.1.3 Conclusion

Malgré la relative « jeunesse » de ces résultats, cette étude exhibe quand même des caractéristiques intéressantes quant à l'utilisation de l'environnement PM^2 pour l'implantation de régulateurs de charge. En effet, d'une part les résultats obtenus vont dans le sens de ceux obtenus par Yves Denneulin à propos du TSP parallèle (efficacité des LRPC) et d'autre part l'implantation des agents locaux sous forme de processus légers montrent bien combien il est aisé, avec PM^2, d'« incorporer » un mécanisme de régulation de charge dans une application pré-existante.

7.4 Conclusion

Dans ce chapitre, nous avons présenté trois applications développées avec la plate-forme PM^2 qui confirment, d'abord que le modèle de programmation proposé convient bien à la conception d'applications parallèles irrégulières, ensuite que les performances globales obtenues sont tout à fait satisfaisantes :

- Yves Denneulin a développé une application d'optimisation combinatoire (TSP) dont l'implantation s'inscrit tout à fait dans la « philosophie » préconisée par l'environnement PM^2, qui incite à utiliser massivement les processus légers. Bien que la pertinence de cette philosophie ait déjà été vérifiée par des mesures de performances « brutes », elle n'en demandait pas moins à être confirmée par une évaluation dans une application réelle. [10] Avec cette application, Yves a donc montré qu'il était possible de créer beaucoup de

10. Il faut bien avouer qu'il était légitime d'avoir quelques petits doutes tout de même !

processus légers (un million de créations sur 300 secondes d'exécution) tout en obtenant des performances plus que raisonnables, ce qui prouve la pertinence de notre approche.

- Christian Perez, après avoir effectué une étude destinée à tester l'intérêt des processus légers dans les programmes à parallélisme de données, a proposé d'utiliser les processus légers pour implanter la notion de *processeur virtuel* d'HPF. L'analyse de cette approche a montré qu'elle constitue un bon compromis entre facilité de mise en œuvre et efficacité d'exécution, tout en exhibant des qualités naturelles de virtualisation de l'architecture. Ces propriétés ont ensuite été vérifiées à l'aide d'un premier prototype appliqué à la résolution de systèmes linéaires de très grande taille (méthode d'élimination de Gauss). Les résultats obtenus confirment le fait qu'une plate-forme à base de processus légers comme PM^2 peut répondre efficacement aux exigences des programmes à parallélisme de données sur architectures distribuées. Une fonctionnalité comme la migration de processus légers semble dans ce cadre promise à un bel avenir.

- Nordine Melab s'est intéressé à la conception d'un régulateur de charge basé sur une politique de récolte d'informations adaptative. L'implantation de ce régulateur, nommé ALBA, est basée sur l'utilisation de processus légers (*agent locaux*) dans chacun des modules d'une application, s'exécutant concurremment (et de façon transparente) avec les traitements de l'application proprement dite. Une première évaluation de ce régulateur a été effectuée avec une application résolvant le problème du jeu de taquin par une version parallèle d'IDA*, basée elle aussi sur l'utilisation de processus légers pour l'exploration de l'espace de recherche. Les résultats obtenus affichent des performances plus qu'honorables pour une souplesse d'utilisation remarquable, puisque l'intégration d'ALBA dans une application distribuée initialement non-régulée ne demande que des modifications mineures.

Si ces trois applications sont assez représentatives des domaines d'utilisation « de prédilection » de l'environnement PM^2, il est important de noter que d'autres applications, dans des domaines parfois éloignés, ont également été développées au-dessus de PM^2. En particulier, et comme le projet PVC en son temps, l'environnement PM^2 a servi de support exécutif pour des applications parallèles à objets. Au sein du projet SLOOP (Nice), l'utilisation de PM^2 pour une extension parallèle de C++ a été évaluée [7], ce qui a mené à la réalisation d'une première maquette par Nathalie Furmento. Dans ce cadre, les fonctionnalités de migration de processus légers fournies par PM^2 ont été exploitées pour mettre en œuvre, au niveau du langage, des mécanismes de migration d'*acteurs* communicants. À Lille, Chrystel Grenot a proposé une évolution du langage BOX [81], nommée TOSCA [83], dont l'implantation s'appuie sur l'environnement PM^2. Dans cette implantation, les appels de méthodes sur les objets distants sont directement supportés par les mécanismes de LRPC fournis par PM^2. Enfin, dans le cadre du projet OSACA (*Open Software Architecture for Cooperative Applications*) auquel nous avons participé, nous avons défini un modèle de développement des applications coopératives basé sur une notion évoluée de *groupe* de processus. L'implantation de la plate-forme associée — nommée GROSACA — est en cours de réalisation au-dessus de PM^2 [120].

Chapitre 8

Conclusions et Perspectives

Les travaux présentés dans cette thèse se sont déroulés au sein de l'équipe GOAL du Laboratoire d'Informatique Fondamentale de Lille dans le cadre du projet ESPACE (*Execution Support for Parallel Applications in high performance Computing Environments*). L'objectif général du projet ESPACE est de définir un cadre méthodologique pour la programmation des applications parallèles irrégulières sur architectures distribuées ainsi qu'un environnement d'exécution complet, portable et efficace pour ces applications.

La difficulté de conception d'un tel environnement est double. D'une part elle intervient au niveau méthodologique où il faut proposer un modèle de programmation des applications irrégulières à la fois simple et adapté aux contraintes d'une architecture distribuée. D'autre part elle intervient au niveau technique où il faut respecter les contraintes de portabilité et d'efficacité de la réalisation auxquelles s'ajoutent celles posées par les applications visées (parallélisme massif, imprévisible et potentiellement de grain très fin).

Pour répondre à ces exigences, l'environnement ESPACE est structuré en trois couches principales qui sont respectivement chargées d'assurer le *support d'exécution*, l'*observation* et l'*équilibrage de charge* des applications. L'objectif de cette thèse était de définir et réaliser la base de l'environnement ESPACE, c'est-à-dire son *support exécutif*.

Dans un premier temps, nous résumons les propositions de la thèse dans ce cadre. Puis, dans un second temps, nous examinons les principales perspectives ouvertes par ce travail.

8.1 Travaux réalisés

La contribution de cette thèse se traduit par la proposition d'un modèle de conception des applications parallèles irrégulières et par la réalisation d'une plate-forme portable et efficace d'expérimentation de ce modèle. Ces deux aspects forment un environnement de programmation distribué nommé PM^2 (*Parallel Multithreaded Machine*).

8.1.1 Cadre méthodologique

Partant de l'objectif ambitieux du projet ESPACE, qui est de fournir un environnement permettant d'exprimer simplement le parallélisme intrinsèque à une application de façon transparente à l'architecture, autrement dit de *virtualiser* l'architecture, nous avons proposé un modèle de programmation et d'exécution des applications parallèles basé sur l'utilisation des *processus légers* pour le support des activités parallèles des applications. Ce modèle

s'articule autour de trois concepts majeurs :

- En proposant une décomposition parallèle des applications à l'aide des mécanismes d'*appel de procédure à distance léger* et de *clonage léger*, le modèle de programmation PM^2 permet au programmeur de se focaliser sur l'expression du parallélisme plutôt que sur la gestion des activités concurrentes. Nous avons montré les multiples avantages de cette approche en matière de facilité d'utilisation, d'efficacité des mécanismes mis en œuvre à l'exécution et surtout de **virtualisation des ressources**.

- À l'exécution, une application conçue au-dessus de PM^2 se compose d'un ensemble distribué de *processus légers* s'exécutant de manière concurrente. Nous avons montré les nombreux avantages liés à l'utilisation de ces derniers pour le support des applications parallèles. En outre, et contrairement à la plupart des environnements à base de processus légers existants, notre approche ne limite pas leur utilisation à l'exploitation du recouvrement des communications par des calculs, mais exploite pleinement le côté « processeur virtuel » conféré par leur modèle. Sur chaque nœud de l'architecture, ces processus légers sont regroupés dans un *module* au sein duquel un ordonnanceur préemptif (avec gestion des priorités) garantit une progression « équitable » de tous les processus. Nous avons montré l'intérêt de cette **concurrence** intra-module pour la virtualisation de l'architecture, mais surtout pour l'équilibrage de charge dynamique en favorisant le maintien d'un degré de parallélisme élevé au sein des applications, donc en facilitant le travail d'un régulateur par migration de processus.

- En fournissant des fonctionnalités permettant la *migration* des processus légers d'un module à un autre (et donc d'un nœud à un autre) pendant leur exécution, l'environnement PM^2 permet non seulement la conception remarquablement souple d'ordonnanceurs globaux (régulateurs de charge, etc.) de toutes sortes, mais il apporte une fonctionnalité majeure — la **mobilité** des activités — dans une démarche de virtualisation réaliste de l'architecture. Nous avons montré l'intérêt de ce mécanisme pour l'exploitation efficace des architectures distribuées en présence d'un parallélisme irrégulier et avons constaté la remarquable simplicité de son utilisation.

La conjonction de ces trois concepts rend possible la conception d'applications complètement indépendantes de l'architecture s'exécutant efficacement en contexte distribué, modulo l'utilisation d'une couche logicielle (typiquement un ordonnanceur global de processus) chargée de la répartition des processus dans les différents modules.

L'utilisation de l'environnement PM^2 dans divers projets de recherche, notamment grâce aux collaborations mises en place lors de l'opération inter-PRC STRATAGÈME, a permis de vérifier la pertinence et l'intérêt de ce modèle pour la conception portable d'applications irrégulières parallèles. En particulier, les applications développées par Yves Denneulin (LIFL), Bertrand Le Cun (PR*i*SM) ou Christian Perez (LIP) montrent bien les avantages obtenus par une *virtualisation* de l'architecture à l'aide de processus légers tant au niveau de la conception qu'au niveau de l'exécution.

8.1.2 Réalisation

L'environnement PM^2, comme beaucoup d'environnements distribués à base de processus légers, est implanté au-dessus d'une bibliothèque de communication et d'une bibliothèque de

gestion de processus légers. Cependant, alors qu'une solution « classique » a été adoptée du point de vue des communications (*i.e.* PVM), l'originalité de l'implantation de PM^2 réside dans l'utilisation d'une bibliothèque de processus légers spécifique — nommée MARCEL — conçue et développée dans le cadre de cette thèse. Ces choix ont été guidés d'une part par des besoins « standard » de l'environnement PM^2 en matière de communications (envoi de messages point-à-point) et d'autre part par des exigences très pointues en matière de gestion des processus légers (migration, efficacité des commutations, etc.).

La bibliothèque MARCEL est une bibliothèque de niveau utilisateur dont l'interface constitue un sous-ensemble des fonctionnalités de POSIX-threads augmenté de fonctionnalités évoluées nécessaires à la réalisation de mécanismes tels que la migration de processus légers. La vocation dédiée au calcul scientifique de cette bibliothèque nous a permis d'une part d'alléger l'interface POSIX pour ne retenir que les fonctionnalités potentiellement utiles pour l'environnement PM^2 et d'autre part de privilégier l'efficacité des mécanismes intensivement sollicités dans ce cadre (commutations en temps constant, allocations optimisées). Actuellement disponible sur six architectures, la bibliothèque MARCEL offre un bon niveau de portabilité car son portage sur une nouvelle architecture ne repose que sur la disponibilité du compilateur `gcc` et sur l'adaptation d'une faible portion du code source. Dans ce domaine, nous avons montré comment il était possible de réaliser des mécanismes tels que la migration, le clonage ou l'extension de pile sur la base d'un outil unique — l'*hibernation* de processus léger — de manière à la fois portable et efficace.

Grâce à la portabilité de la bibliothèque MARCEL et à la disponibilité de PVM, l'environnement PM^2 est déjà opérationnel sur six architectures : Sparc/SunOS, Sparc/Solaris, Alpha/OSF-1, PC/Linux, PowerPC/Aix et Mips/Irix.

Différentes mesures de performances ont été effectuées pour évaluer l'efficacité « brute » des mécanismes fournis par l'environnement PM^2 :

- Nous avons comparé les temps d'exécution des mécanismes de PM^2 sollicitant des communications (LRPC, migrations) par rapport aux temps d'exécution des mécanismes de communication natifs sous-jacents (PVM). Les résultats obtenus montrent les bonnes performances générales de cette implantation. Dans le cas des LRPC par exemple, ces mesures montrent que le surcoût relatif de PM^2 par rapport à PVM est très faible, même pour des petits messages.

- Nous avons comparé les performances de MARCEL avec d'autres gestionnaires de processus légers. Les résultats obtenus sont remarquables et placent cette bibliothèque au niveau des toutes meilleures du point de vue de l'efficacité. Ils valident les nombreux choix que nous avons effectués lors de sa conception.

Le bilan de ces premières constatations indique que l'implantation de PM^2 représente un intéressant équilibre entre la richesse des **fonctionnalités** fournies, la **portabilité** de la plate-forme et l'**efficacité** des mécanismes mis en œuvre.

L'utilisation de l'environnement PM^2 dans les divers projets de recherche mentionnés auparavant (optimisation combinatoire, parallélisme de données, etc.) a permis de mettre en évidence à la fois les bonnes performances des applications développées et leur indépendance vis-à-vis de l'architecture sous-jacente. Ces résultats montrent que la **virtualisation** de l'architecture à l'aide des **processus légers** est une approche réaliste et efficace. Ils sont une première validation de notre approche. Dans ce domaine, nous avons mené des expériences

visant à mesurer l'impact d'une utilisation massive des processus légers dans les applications. Les résultats obtenus montrent que cette politique n'est pas coûteuse et confirment la pertinence de notre approche pour le support des applications massivement parallèles.

8.2 Perspectives

Les travaux réalisés dans le cadre de cette thèse avaient pour objectif de proposer la définition et la réalisation d'un support exécutif destiné à constituer la base du projet ESPACE. Notre contribution dans ce domaine, concrétisée par la proposition de l'environnement PM^2, a permis d'ouvrir de multiples perspectives de recherche dans trois domaines principaux : les collaborations avec d'autres équipes de recherche, l'évolution du modèle de programmation et l'extension de la réalisation actuelle à de nouvelles technologies.

8.2.1 Collaborations

Nous avons relaté, à plusieurs reprises, les nombreuses collaborations que nous avons menées avec différentes équipes de recherche françaises. Établies, pour la plupart, avec des équipes potentiellement « clientes » de l'environnement PM^2, ces collaborations ont eu une influence considérable sur certains choix de conception de notre environnement en nous aidant à mieux cerner les différentes contraintes et les multiples besoins des applications parallèles irrégulières. Naturellement, le maintien et la création de nouvelles collaborations constituent un axe que nous privilégions :

- Le support des langages à parallélisme de données capables de manipuler des structures de données irrégulières semble être un domaine d'utilisation très prometteur pour l'environnement PM^2.

  Nous l'avons vu, l'équipe PARADIGME (L. Bougé, LIP) s'intéresse à la conception d'outils permettant l'exécution efficace d'applications irrégulières développées dans le langage HPF. Dans ce cadre, les processus légers semblent être de très bons candidats au support des *processeurs virtuels* et les fonctionnalités de migration offertes par l'environnement PM^2 revêtent une importance particulière. Notre collaboration avec l'équipe PARADIGME nous a déjà permis d'améliorer le protocole de migration de processus légers (transport de données globales annexes) et de réaliser une gestion de groupes de processus légers distribués dotés de barrières de synchronisation. Dans un avenir proche, des travaux communs devraient nous permettrent d'améliorer encore le mécanisme de migration de processus, en intégrant une notion d'*espace de tas privé* pour chaque processus léger.

  L'équipe WEST (J.L. Dekeyser, LIFL), quant à elle, adopte une démarche basée sur la proposition d'un nouveau langage — nommé IDOLE — spécialement conçu autour d'un modèle de programmation « data-parallèle irrégulier » [49]. Ce modèle définit des machines virtuelles irrégulières et dynamiques et le modèle d'exécution associé repose sur la mise en œuvre de *processeurs virtuels* sur la machine cible. Là encore, les processus légers représentent une cible naturelle de ce processus de virtualisation. Une collaboration avec l'équipe WEST s'est donc établie autour de l'utilisation de l'environnement PM^2 en tant que support pour la compilation de ce langage. Dans ce cadre, Julien Soula a débuté une thèse dont l'objectif est la conception et la mise en œuvre d'un support

d'exécution intermédiaire intégrant des fonctionnalités spécialement adaptées au langage IDOLE (systèmes d'allocations dynamiques, protocoles de synchronisation, équilibrage de charge, etc.). Des travaux communs sont en cours pour évaluer (et faire évoluer) le mécanisme de *clonage léger* compte tenu des exigences des applications visées.

- Notre participation au sous-groupe « *Support d'Exécution* » du thème 1.2 du PRC PRS (EXEC) nous a permis d'établir des contacts enrichissants avec la communauté Française du domaine. Plus particulièrement, elle nous a rapproché des membres du projet APACHE (B. Plateau, LMC) dont certains travaux de recherche sont proches des nôtres. En particulier, un rapport de synthèse commun est en cours d'élaboration avec l'équipe de Jacques Briat, qui s'intéresse également à la conception de supports exécutifs à base de processus légers pour applications irrégulières. Cette collaboration devrait aboutir, dans un avenir proche, à une réflexion commune sur la définition d'une « couche de portabilité » fournissant une interface minimale à des fonctionnalités de communications *thread-aware*.

- Une collaboration avec le laboratoire d'Analyse Numérique et d'Optimisation (C. Brezinski, Lille) a été établie depuis peu et son objectif réside dans l'étude et la réalisation de la parallélisation de méthodes hybrides de résolution de systèmes linéaires. Il a été montré théoriquement que ces méthodes, consistant en une combinaison de solutions approximées arbitraires, convergent plus vite que les méthodes classiques [16]. La parallélisation de ces méthodes à l'aide de l'environnement PM^2 semble prometteuse car elle autoriserait l'évaluation concurrente d'un très grand nombre de solutions approximées tout en offrant la possibilité d'équilibrer ces évaluations dynamiquement sur l'architecture.

8.2.2 Modèle de programmation et d'exécution

Un des principaux avantages du modèle de conception des applications parallèles proposé dans cette thèse repose sur le petit nombre de fonctionnalités fournies, ce qui simplifie grandement la tâche du programmeur. En contrepartie, la difficulté de conception de l'interface de ces fonctionnalités s'en trouve fortement augmentée, à cause de la « généricité » qu'il faut nécessairement préserver. De ce point de vue, les choses s'effectuent souvent empiriquement et se stabilisent avec le nombre d'applications traitées. C'est pourquoi, grâce aux retombées des multiples expérimentations de PM^2 (parfois dans d'autres équipes de recherche), nous entrevoyons deux évolutions de son interface dans un avenir proche :

clonage léger Comme nous l'avons signalé, le mécanisme de *clonage léger* est de conception récente et son expérimentation dans des applications « réelles » n'a pas encore été effectuée. La situation est en passe d'évoluer avec les collaborations que nous menons avec les équipes de Jean-Luc Dekeyser et de Luc Bougé. En effet, de par sa nature, ce mécanisme s'avère assez bien adapté à la décomposition des programmes à parallélisme de données. Une étude précise des conditions typiques de son utilisation devraient conduire à un enrichissement de son interface, notamment en ce qui concerne les mouvements de données accompagnant les phases de *séparation* et de *fusion*.

objets partagés Dans les applications nécessitant des schémas de communication sortant du cadre strict des transmissions de données de père à fils lors d'un LRPC synchrone, telles

que des régulateurs de charge distribués par exemple, l'utilisation des LRPC asynchrones devient nécessaire. Dans ce cas, les échanges d'informations entre processus doivent s'effectuer via des zones de mémoires bien définies, comme c'est le cas dans l'exemple présenté page 160. Bien qu'aucun support sémantique particulier ne soit fourni par l'interface PM^2 dans ce domaine, il est parfois nécessaire d'utiliser des mécanismes de désignation équivalents aux *pointeurs globaux* de Nexus pour implanter ces schémas de communication. Dans cette optique, nous pensons qu'une intégration dans l'interface d'une notion élémentaire d'*objet partagé* faciliterait beaucoup la conception de telles applications.

La richesse des fonctionnalités fournies par l'environnement PM^2 s'explique par une volonté de permettre une virtualisation de l'architecture la plus réaliste possible. Bien que nous ayons pu constater l'ampleur de sa contribution dans ce domaine, l'environnement PM^2 ne fournit pas encore de solutions tout à fait satisfaisantes dans les domaines suivants :

va-et-vient de processus Actuellement, le nombre maximum de « processeurs virtuels » offerts par l'environnement PM^2 est limité[1] par la mémoire primaire (physique) des nœuds de l'architecture sous-jacente. Cette caractéristiques est due à l'incompatibilité des mécanismes de *va-et-vient* des systèmes d'exploitation avec l'implantation de PM^2. Ce problème ainsi que les solutions que nous y envisageons sont précisément détaillés en section 5.6.1.

migration La migration de processus légers est un mécanisme clé en ce qui concerne l'exécution efficace des applications fortement irrégulières en contexte distribué. Cependant, sa mise en œuvre sans l'assistance d'un compilateur reste très difficile. Le mécanisme proposé par PM^2 n'échappe pas à la règle et, en autorisant une utilisation à la fois souple et efficace, introduit des limitations sévères en ce qui concerne la manipulation des pointeurs (cf. sections 5.4.5.3 et 5.4.5.4). À cause de cela, certaines applications peuvent contenir une majorité de processus légers constamment dans un état *non-migrable*, ce qui peut nuire au fonctionnement d'un ordonnanceur global par migration par exemple. Pour éviter ces problèmes (ou du moins une partie), certains environnements n'autorisent que l'automigration des processus (Ariadne [115]) ou pré-allouent les processus légers sur tous les nœuds (UPVM [102]). La quête d'une solution à cet épineux problème demeure un thème de recherche nouveau [44]. Nous comptons poursuivre nos travaux dans ce domaine en évaluant d'un côté des techniques de compilation et d'un autre côté des techniques d'analyse de fichiers objets qui pourraient nous fournir les informations (dont nous manquons pour le moment) concernant la manipulation des pointeurs par le processeur.

8.2.3 Réalisation

La réalisation actuelle de la plate-forme PM^2 est disponible sur six architectures. Ces différents portages sont presque tous (sauf sur IBM SP2) basés sur la librairie PVM du domaine public (dont nous avons évoqué la faible efficacité) et concernent tous des machines dont les

1. Au-delà de cette limite, le système peut continuer de fonctionner, mais une dégradation importante des performances risque alors d'apparaître.

nœuds sont monoprocesseurs. Nous comptons élargir le parc des architectures supportées en envisageant l'intégration des technologies suivantes :

réseaux haut-débit Dans le cadre d'une collaboration industrielle avec la société Newbridge (un des leaders sur le marché ATM), le laboratoire de Lille s'est équipé de commutateurs ATM (2 clusters de stations, 12 voies par cluster) et un contrat nous permet d'utiliser ces commutateurs via une interface de programmation (API) propriétaire. Dans ce cadre, le portage de l'environnement PM^2 sur ce type de réseau a été retenu en tant que projet principal. Nous menons donc actuellement diverses expérimentations destinées à mettre en évidence le bon compromis *fonctionnalité/efficacité* qu'il faut adopter pour un tel réseau. Si ces expérimentations confirment qu'ATM est un bon support pour le parallélisme, alors la définition d'une « couche minimale de portabilité » sera entreprise, de manière à favoriser les portages ultérieurs sur d'autres types de réseaux (Myrinet par exemple).

nœuds multiprocesseurs L'implantation actuelle de l'ordonnanceur des processus de la bibliothèque MARCEL ne permet pas un parallélisme réel entre les exécutions des processus légers, même lorsque la machine sous-jacente (*i.e.* le nœud) est multiprocesseurs. Cette caractéristique, due à l'exploitation d'une seule entité d'exécution du noyau, n'est pas acceptable sur une machine multiprocesseurs : s'il veut pleinement profiter des ressources de la machine, le programmeur est obligé de placer plusieurs *modules* sur un même nœud et effectuer un « macro-ordonnancement » entre ces modules en utilisant la migration de processus légers.[2] La seule solution satisfaisante à ce problème est donc de modifier l'implantation de l'ordonnanceur MARCEL de façon à lui donner la possibilité de placer des processus légers d'un même module sur différents processeurs. Une solution bien connue est de multiplexer l'ordonnancement des processus légers au-dessus d'un *pool* de processus lourds (PRESTO [11], Ariadne [115]). Une autre solution, de mise en œuvre plus directe, consiste à effectuer ce multiplexage au-dessus d'un *pool* de processus de poids moyen, qui sont disponibles dans la plupart des systèmes d'exploitations pour machines multiprocesseurs actuelles. Un projet d'ingénieur sera donné prochainement pour intégrer cette caractéristique dans l'environnement PM^2.

2. Avouons que c'est dommage en présence d'une mémoire commune aux processeurs !

Bibliographie

[1] AGHA, G. *Actors. A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[2] ANDERSON, T., BERSHAD, B., LAZOWSKA, E., AND LEVY, H. Scheduler Activations: Efficient kernel support for the user-level managment of parallelism. In *Proc. of the Thirteenth ACM Symposium on Operating Systems Principles* (oct 1991), vol. 10, pp. 95–105.

[3] ARNOLD, K., AND GOSLING, J. *The Java Programming Language.* The Java Series. SunSoft Press, 1996.

[4] BACH, M. *The design of the UNIX operating system.* Prentice-Hall International Editions, 1986.

[5] BAL, H., KAASHOEK, F., AND TANENBAUM, A. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering 18*, 3 (Mar 1992), 190–205.

[6] BARTON-DAVIS, P., MACNAMEE, D., VASWANI, R., AND LAZOWSKA, E. Adding scheduler activations to mach 3.0. DCSE 92-08-03, University of Washington, 1992.

[7] BAUDE, F., FURMENTO, N., CAROMEL, D., NAMYST, R., GEIB, J., AND MEHAUT, J. C++// on top of PM^2 via Schooner. In *International Conference Stratagem'96* (Sophia Antipolis, Jul 1996).

[8] BERMOND, J., CAROMEL, D., AND MUSSI, P. Simulation within a Parallel Object-Oriented Language. In *Second annual joint conference on Information Sciences (JCIS'95).* (1995).

[9] BERNARD, P., PLATEAU, B., AND TRYSTRAM, D. Using threads for developing applications: Molecular dynamics as a case study. In *Parallel Numerics 96* (Slovenia, Sep 1996).

[10] BERSHAD, B., ANDERSON, T., LAZOWSAK, E., AND LEVY, H. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems 8*, 1 (Feb. 1990), 37–55.

[11] BERSHAD, B., LAZOWSKA, E., AND LEVY, H. Presto: A system for object-oriented programming. *Software Practice and Expérience 18*, 8 (1988), 713–732.

[12] BIRELL, A., AND NELSON, B. Implementing Remote Procedure Calls. Tech. Rep. CSL-83-7, XEROX, 1983.

[13] BLUMOFE, R., JOERG, C., KUSZMAUL, B., LEISERSON, C., RANDALL, K., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. In *PPOP'95 proceedings ACM SIGPLAN* (Santa Barbara, 1995).

[14] BLUMOFE, R., AND LEISERSON, C. Scheduling Multithreaded Computations by Work Stealing. In *35th Annual Symposium on Foundations of Computer Science (FOCS'94)* (New-Mexico, Nov. 1994), pp. 356–368.

[15] BRANDES, T. *Adaptor (HPF compilation system), developped at GMD-SCAI.* Available at URL `http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html`.

[16] BREZINSKI, C., AND REDIVO ZAGLIA, M. Hybrid procedures for solving linear systems. *Numerische Mathematik 1*, 67 (1994), 1–19.

[17] BRIAT, J., GAUTIER, T., AND ROCH, J. On-line scheduling. In *European School of Computer Science: Parallel Programming Environnement for High Performance Computing* (Alpe d'Huez France, Apr. 1996), pp. 95–108.

[18] BRUNIE, L., AND LEFEVRE, L. DOSMOS: a distributed shared memory based on pvm. In *First European PVM users'group meeting* (Italy, Oct 1994), Rome university.

[19] BUBECK, T. Eine Systemumebung zum vereilen funktionalen Rechnen. Tech. Rep. WSI-93-8, Universität Tübingen, Germany, Aug. 1993.

[20] BUBECK, T., AND ROSENSTIEL, W. Verteiltes Rechnen mit DTS (Distributed Thread System). In *Proc. '94 SIPAR-Workshop on Parallel and Distributed Computing* (Suisse, Oct. 1994), M. Aguilar, Ed., Fribourg, pp. 65–68.

[21] BUTLER, R., AND LUSK, E. Monitors, messages, and clusters: the P4 parallel programming system. Tech. Rep. MCS-P362-0493, Argonne National Laboratory, Argonne, IL, USA, 1993.

[22] CALMETTES, C., AND GALL, F. L. Solving markov systems using matrix reduction; implementation on a transputer system. In *Tempus Jep 2011 - IMPACT - Distributed Control* (Prague, may 1994), J. Bilek, Ed., Czech Technical University in Prague, pp. 85–90.

[23] CARLISLE, M., AND ROGERS, A. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM Symposium on Principles and Practices of Parallel Programming* (1995).

[24] CARRIERO, N., AND GELERNTER, D. *How to Write Parallel Programs: A first course.* MIT Press, 1990.

[25] CASAS, J., CLARK, D., KONURU, R., OTTO, S., PROUTY, R., AND WALPOLE, J. MPVM: A Migration Transparent Version of PVM. *Computing Systems 8*, 2 (1993), 171–216.

[26] CASAS, J., KONURU, R., OTTO, S., PROUTY, R., AND WALPOLE, J. Adaptative load migration systems for PVM. In *Proceedings of Supercomputing* (Washington D.C., November 1994), ACM/IEEE, pp. 390–399.

[27] CHANDY, K., AND KESSELMAN, C. CC++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming* (1993), MIT Press.

[28] CHANDY, K., AND KESSELMAN, C. Compositional C++: Compositional parallel programming. In *Proc. Fifth Int'l Workshop on Parallel Languages and Compilers* (1993), Springer-Verlag.

[29] CHARLES, H., LI, J., AND MIGUET, S. 3D image processing on a distributed memory parallel computer. In *Biomedical Image Processing and biomedical vizualisation* (San José California, Feb. 1993), A. Godolf, Ed., pp. 379–390.

[30] CHRISTALLER, M. Athapascan-0a: A control parallelism approach on top of PVM. In *Proceedings of the 1994 PVM Users' Group Meeting* (Oak Ridge, Tennessee, 1994).

[31] CHRISTALLER, M. *Athapascan-0: vers un support exécutif pour applications parallèles irrégulières efficacement portables.* PhD thesis, Université Joseph Fourier, Grenoble I, Nov 1996.

[32] CHRISTALLER, M., BRIAT, J., AND RIVIÈRE, M. Athapascan-0 : Concepts structurants simples pour une programmation parallèle efficace. *Calculateurs Parallèles*, 7(2) (1995), 173–196.

[33] CHRISTALLER, M., BRIAT, J., AND RIVIÈRE, M. Athapascan-0 : Vers une expression du parallélisme à l'aide de décompositions en procédures parallèles et procédures barrière. In *Actes des 7ièmes Rencontres Francophones du Parallèlisme (Renpar7)* (Mons, 1995).

[34] CHRISTALLER, M., CASTANEDA-RETIZ, M., AND GAUTIER, T. Control parallelism on top of PVM: The Athapascan environment. In *EuroPVM'95 proceedings* (Lyon, 1995), Hermès.

[35] COLIN, J. *DTMS: Un environnement pour la programmation distribuée à grain indéterminé.* PhD thesis, Université de Mons-Hainaut, 1995.

[36] COLLINS, G., AND LOOS, R. Specification and index of SAC-2 algorithms. Tech. Rep. WSI-90-4, Universität Tübingen, Germany, 1990.

[37] CONWAY, M. Design of separatable transition diagram compiler. *Communication of ACM 7* (Jul 1963), 396–408.

[38] COURTRAI, L. *Les Composants Actifs de Communication: Outils pour la conception et l'implantation de langages parallèles à objets actifs pour machines MIMD.* PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Oct. 1992.

[39] COURTRAI, L., DELATTRE, E., AND GEIB, J. A Server Based Architecture to Support Object-Oriented Languages on Multicomputers (v 2.0). Tech. Rep. ERA-95, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, Apr. 1991.

[40] COURTRAI, L., ROOS, J., GEIB, J., AND MÉHAUT, J. Communicating active components: an environment for concurrent applications on parallel machines. In *Proceedings of EUROMICRO Hardware and Software Design Automation* (1992).

[41] COURTRAI, L., ROOS, J., GEIB, J., AND MÉHAUT, J. Communicating Active Components: an Environment for Concurrent Applications on Parallel Machines. In *Proc. of EUROMICRO Hardware and Software Design Automation, published in Microprocessing and Microprogramming* (Sept. 1992), vol. 35(1-5), pp. 47–54.

[42] COURTRAI, L., ROOS, J.-F., GEIB, J.-M., AND MÉHAUT, J.-F. Communicating Active Components: an Environment for Concurrent Applications on Parallel Machines. *Proceedings of EUROMICRO Hardware and Software Design Automation, published in Microprocessing and Microprogramming 35*, 1-5 (Sept. 1992), 47–54.

[43] COURTRAI, L., ROOS, J.-F., AND MÉHAUT, J.-F. Support of an Actor Environment on Distributed Architectures. In *Proceedings of SUG'92 Tenth Annual Sun User Group Conference* (San Jose, Dec. 1992).

[44] CRONK, D., HAINES, M., AND MEHROTRA, P. Thread migration in the presence of pointers. In *Proceedings of Thirtieth Hawaii Internation Conference on Systems Sciences (HICSS'97)* (Jan 97). To appear.

[45] CUN, B. L. *Structures de données parallèles et optimisation combinatoire.* PhD thesis, Université Pierre et Marie Curie – Paris VI, 1995.

[46] CUN, B. L., ROUCAIROL, C., CUNG, V., AND MAUTOR, T. BOB: A unified platform for implementing branch-and-bound like algorithms. Tech. Rep. 95-16, PRISM, Versailles, 1995.

[47] CUNG, V. *Contribution à l'Algorithmique Non Numérique Parallèle : Exploration d'Espaces de Recherche.* PhD thesis, Université Pierre et Marie Curie – Paris VI, Avril 1994.

[48] CUSTER, H. *Inside Windows NT.* Microsoft Press, 1993.

[49] DEKEYSER, J., AND MARQUET, P. Supporting irregular and dynamic computations in data-parallel languages. In *The Data Parallel Programming Model* (Les Menuires, Mars 1996), pp. 197–219. Lectures Notes in Computer Science, Tutorial 1132.

[50] DELAMARRE, D. *Étude et conception d'algorithmes parallèles d'optimisation combinatoire discrète approchée; implantation sur architecture MIMD.* PhD thesis, Université d'Orléans, Décembre 1994.

[51] DENNEULIN, Y., GEIB, J., GRANSART, C., HEMERY, F., MÉHAUT, J., ROOS, J., TALBI, E., GAUTIER, T., ROCH, J., AND VERMEERBEREN, A. *Parallélisme et applications irrégulières.* Hermes, 1995, ch. Structures de Données Irrégulières et régulation de charge, pp. 203–239.

[52] DENNEULIN, Y., GEIB, J., AND MÉHAUT, J. Solving irregular problems in Parallel. In *Parallel Optimization Colloquium 96* (Versailles France, Mar. 1996), pp. 75–83.

[53] DENNEULIN, Y., AND NAMYST, R. PM2: Parallel Multithreaded Machine. Un support d'exécution pour applications irrégulières. In *7èmes Rencontres du Parallélisme* (Mons Belgique, May 1995), pp. 208–212.

[54] DENNEULIN, Y., NAMYST, R., GEIB, J., AND MÉHAUT, J. LBMP: Load-Balancing with Migration directed by Priorities. In *European School of Computer Science: Parallel Programming Environnement for High Performance Computing* (Alpe d'Huez France, Apr. 1996), pp. 115–118.

[55] DIJKSTRA, E. *Co-operating Sequential Processes in Programming Languages*, F. Genuys ed. London: Academic Press, 1965.

[56] DISZ, T., PAPKA, M., PELLEGRINO, M., AND STEVENS, R. Sharing visualization experiences among remote virtual environments. In *Int. Workshop on High Performance Computing for Computer Graphics and Visualization* (1995), Springer-Verlag, pp. 217–237.

[57] DOUGLIS, F. Process migration in the Sprite operating system. Tech. Rep. UCB/CSD 87/343, University of California, Berkeley, February 1987.

[58] DOWAJI, S., AND ROUCAIROL, C. Influence of priority of tasks on load balancing strategies for distributed branch-and-bound algorithms. In *IPPS'95, Workshop on Solving Irregular Problems on Distributed Memory Machines* (Santa Barbara, 1995).

[59] DREIER, B., AND ZAHN, M. RThreads — a Uniform Interface for Parallel and Distributed Programming. In *Proceedings of the Second International Conference on Massively Parallel Computing Systems (MPCS'96)* (Ischia, Italy, May 1996), pp. 557–561.

[60] DUMOULIN, C. *DREAM: une mémoire partagée distribuée à cohérence programmable.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, Jan 1997. À paraître.

[61] ELLEUCH, A. *Migration de Processus dans les systèmes massivement parallèles.* PhD thesis, Université de Grenoble, Novembre 1994.

[62] ENGLER, D., ANDREWS, G., AND LOWENTHAL, D. Efficient Support for Fine-Grain Parallelism. Tech. rep., University of Arizona, 1993.

[63] FACQ, L., AND ROMAN, J. *Parallélisme et Applications Irrégulières*, Hermès ed. G. Authié and al., 1995, ch. Algèbre Linéaire Creuse.

[64] FAGOT, A., AND CHASSIN DE KERGOMMEAUX, J. Optimized record-replay mechanism for rpc-based parallel programming. In *Proc. of IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems* (Ascona Switzerland, Apr. 1994), Birkhaeuser Verlag.

[65] FERRARI, A., AND SUNDERAM, V. TPVM. A Threads-Based Interface and Subsystem for PVM. Tech. Rep. CSTR-940802, University of Virginia & Emory University, USA, August 1994.

[66] FERRARI, A., AND SUNDERAM, V. Multiparadigm distributed computing with TPVM. Tech. Rep. CSTR-951201, University of Virginia & Emory University, USA, 1995.

[67] FERRARI, A., AND SUNDERAM, V. TPVM: Distributed Concurrent Computing with Lightweight Processes. In *Proc. of IEEE High Performance Distributed Computing* (Washington, 1995), D.C., pp. 211–218.

[68] FORD, B., HIBLER, M., AND LEPREAU, J. Notes on thread models in Mach 3.0. Tech. Rep. UUCS-93-012, Department of Computer Science, University of Utah, Apr 1993.

[69] FORD, B., AND LEPREAU, J. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Conference* (January 1994).

[70] FOSTER, I., AND CHANDY, K. Fortran M: A language for modular parallel programming. *Jounal on Parallel and Distributed Computing*, 25(1) (1994).

[71] FOSTER, I., KESSELMAN, C., OLSON, R., AND TUECKE, S. Nexus: An interoperability toolkit for parallel and distributed computer systems. Tech. Rep. ANL/MCS-TM-189, Argonne National Laboratory, USA, 1994.

[72] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The Nexus task-parallel runtime system. In *Proc. 1st Intl Workshop on Parallel Processing* (1994), Tata Mc Graw Hill.

[73] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The Nexus approach to integrate multithreading and communication. Tech. rep., Argonne National Laboratory, USA, 1995.

[74] FOSTER, I., AND OLSON, R. A guide to parallel and distributed programming in nperl. Tech. Rep. `http://www.mcs.anl.gov/nexus/nperl`, Argonne National Laboratory, USA, 1995.

[75] GAUTIER, T., ROCH, J., AND VILLARD, G. Regular versus Irregular Problems and Algorithms. In *Proc. of IRREGULAR'95* (Lyon, Sept. 1995), Springer-Verlag.

[76] GEIB, J., DENNEULIN, Y., NAMYST, R., AND MÉHAUT, J. Processus légers distribués et régulation de charge. In *Actes de l'école d'été sur le Placement et la Régulation de Charge (PRC'96)* (Presqu'île de Giens, Jul 1996).

[77] GEIB, J., GRANSART, C., AND GRENOT, C. Distributed objects in BOX. In *TOOLS'93, Workshop on Distributed Objects and Concurrency* (Versailles, 1993), pp. 195–199.

[78] GEIB, J., HÉMERY, F., MÉHAUT, J., ROOS, J., AND TALBI, E. PVC: un environnement de programmation parallèle avec régulation de charge. *Lettre des Calculateurs Parallèles 7*, 2 (1995), 139–159.

[79] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHECK, R., AND SUNDERAM, V. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, 1994.

[80] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHECK, R., AND SUNDERAM, V. *PVM3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Tennessee, May 1995.

[81] GRANSART, C. *BOX : Un modèle et un langage à objets pour la programmation parallèle et distribuée.* PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Jan. 1995.

[82] GRANSART, C., GRENOT, C., AND COURTRAI, L. Le Projet PVC/BOX Environnement pour la Programmation Parallèle. In *Actes des Journées des Jeunes Chercheurs en Systèmes Informatiques Répartis* (Apr. 1993), DRED MRE-CNRS, IMAG, pp. 131–135.

[83] GRANSART, C., GRENOT, C., AND MERLE, P. Tosca : Support de threads et d'objets pour applications coopératives. In *Actes des Journées des Jeunes Chercheurs en Systèmes Informatiques Répartis* (Rennes, Oct. 1995), IRISA.

[84] GROPP, W., AND LUSK, E. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Tech. Rep. MCS-P342-1193, Argonne National Laboratory, 1994.

[85] GROPP, W., AND LUSK, E. MPICH working note: Creating a new MPICH device using the channel interface. Tech. Rep. ANL/MCS-TM-213, Argonne National Laboratory, 1995.

[86] GROPP, W., LUSK, E., DOSS, N., AND SKELLUM, A. A high performance, portable implementation of the MPI message passing interface standard. Tech. Rep. ANL/MCS-TM-213, Argonne National Laboratory, 1996.

[87] HAFIDI, Z., TALBI, E., AND GEIB, J. MARS: An approach for scheduling parallel applications. In *European School of Computer Science : Parallel Programming Environnement for High Performance Computing* (Alpe d'Huez France, Apr. 1996), pp. 119–122.

[88] HAINES, M., AND BOHM, W. An evaluation of software multithreading in a conventional distributed-memory multiprocessor. In *IEEE Symp. on Parallel and Distributed Processing* (1993), pp. 106–113.

[89] HAINES, M., CRONK, D., AND MEHROTRA, P. On the design of chant: A talking threads package. In *Proc. of Supercomputing'94* (Washington, November 1994), pp. 350–359.

[90] HAINES, M., HESS, B., MEHROTRA, P., ROSENDALE, J., AND ZIMA, H. Runtime support for data parallel tasks. In *Proceedings of The Fifth Symposium on the Frontiers of Massively Parallel Computation* (February 1995), Mc Lean, pp. 432–439.

[91] HAINES, M., MEHROTRA, P., AND CRONK, D. Chant: Lightweight Threads in a Distributed Memory Environment. Tech. rep., ICASE, NASA Langley Research Center, May 1995.

[92] HAINES, M., MEHROTRA, P., AND CRONK, D. Ropes : Support for collective operations among distributed threads. Tech. Rep. 95-36, ICASE, May 1995.

[93] HÉMERY, F. *Etude de la répartition dynamique d'activités sur architectures décentralisées.* PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, June 1994.

[94] HIGH PERFORMANCE FORTRAN FORUM. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, Nov 1994. Version 1.1.

[95] HIGH PERFORMANCE FORTRAN FORUM. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, Oct 1996. Version 2.0.

[96] HÉMERY, F., AND GEIB, J. Simulation pour l'Aide au Placement d'Entités Actives Communicantes. In $5^{ième}$ *Rencontres du Parallélisme* (May 1993), Laboratoire d'Informatique de Brest (équipe Armen), Ed., pp. 195–199.

[97] HOARE, C. Communicating Sequential Processes. *Communications of ACM 21*, 8 (Aug 1978), 666–677.

[98] HSIEH, W., WANG, P., AND WEIHL, W. Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems. In *Proceedings of the Symposium on Principles and Practices of Parallel Programming* (1993), pp. 239–248.

[99] IEEE STANDARDS DEPARTMENT. *1003.4d8 POSIX System Application Program Interface: Threads Extensions [C language]*, 1994.

[100] JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 1*, 6 (February 1988), 109–133.

[101] KOKOSZKO, B. Intégration du modèle data-parallèle irrégulier Idole dans C++. In *RenPar8, 8es Rencontres sur le Parallélisme* (Bordeaux, May 1996).

[102] KONURU, R., CASAS, J., OTTO, S., PROUTY, R., AND WALPOLE, J. A User-Level Process Package for PVM. In *Proceedings of the Scalable High Performance Computing Conference* (Knoxville, Tennessee., May 1994), ACM/IEEE, pp. 48–55.

[103] KONURU, R., OTTO, S., WALPOLE, J., PROUTY, R., AND CASAS, J. A User-Level Process Package for Concurrent Computing. Tech. Rep. 93-016, Oregon Graduate Institute, August 1993.

[104] KORF, R. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence 32*, 27 (Feb 1985), 97–109.

[105] KRAKOWIAK, S., MEYSEMBOURG, M., VAN, H. N., RIVEILL, M., ROISIN, C., AND DE PINA, X. R. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object Oriented Programming 3*, 3 (Sep 1990), 11–22.

[106] KÜCHLIN, W. PARSAC-2: A parallel SAC-2 based on threads. In *Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes: 8th International Conference* (Tokyo, Aug. 1990), S. Sakata, Ed., vol. 508 of *LNCS*, Springer-Verlag, pp. 341–353.

[107] LATTEUX, M. Synchronisation de processus. *RAIRO Informatique 14*, 2 (1980), 103–135.

[108] LEA, R., AND WEIGHTMAN, J. Supporting object oriented languages in a distributed environment: The COOL approach. In *Proceedings of Tools USA* (Santa-Barbara, 1991), p. 13.

[109] LEWIS, B., AND BERG, D. *Threads Primer. A Guide to Multithreaded Programming.* Prentice Hall, 1996.

[110] LI, K., AND HUDAK, P. Memory coherence in shared memory systems. *ACM transactions on Computer Systems 7*, 4 (Nov 1989), 321–359.

[111] MANCHEK, R. Design and implementation of PVM version 3. Master's thesis, University of Tennessee, Knoxville, Tennessee, 1994.

[112] MARION-POTY, V. *Approches Parallèles pour la Squelettisation 3D.* PhD thesis, Ecole Normale Supérieure de Lyon, Dec. 1994.

[113] MARQUES, J. A., BALTER, R., CAHILL, V., GUEDES, P., HARRIS, N., HORN, C., KRAKOWIAK, S., KRAMER, A., SLATERY, J., AND VANDOME, G. Implementing the Comandos architecture. In *Proceedings of the 5th Annual Esprit Conference* (Brussels, Nov 1988), pp. 1140–1157.

[114] MARSH, B., LEBLANC, T., SCOTT, M., AND MARKAROS, E. First-Class User-Level Threads. Tech. rep., Computer Science Department, University of Rochester, USA, 1991.

[115] MASCARENHAS, E., AND REGO, V. Ariadne: Architecture of a portable threads system supporting mobil processes. Tech. Rep. CSD-TR-95-017, Purdue University, March 1995.

[116] MEHROTRA, P., AND HAINES, M. An overview of the Opus language and runtime system. In *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computers* (New York, November 1994).

[117] MELAB, N., DEVESA, N., LECOUFFE, M., AND TOURSEL, B. An Adaptative Load Balancing Algorithm with a Multithreaded Implementation. In *International Conference on Systems Engineering (ICSE'96)* (Las Vegas, Jul 1996).

[118] MELAB, N., DEVESA, N., LECOUFFE, M., AND TOURSEL, B. Adaptative Load Balancing of Irregular Applications. A Case Study : IDA* Applied to the 15-Puzzle Problem. In *Proc. of the 3rd Intl. Workshop IRREGULAR'96* (Santa Barbara, California, Aug 1996), Springer-Verlag LNCS 1117.

[119] MELAB, N., DEVESA, N., LECOUFFE, M., AND TOURSEL, B. An Adaptative Load Information Collection Policy. A Case Study : Load Balancing. In *International Conference on Parallel and Distributed Processing Techniques and Applications* (Sunnyvale, Aug 1996).

[120] MERLE, P., NAMYST, R., GEIB, J.-M., AND MÉHAUT, J.-F. Objects Spaces, Cooperation Spaces and Groups. In *ACM-SIGOPS 94, Sixth SIGOPS European Workshop* (Wadern Germany, Sept. 1994).

[121] MESSAGE PASSING INTERFACE FORUM. *MPI: A Message-Passing Interface Standard*, March 1994. available from netlib2.cs.utk.edu.

[122] MÉHAUT, J., AND NAMYST, R. Deux approches pour la programmation parallèle: PVM et LINDA. In *Actes de l'école d'été sur le Placement et la Régulation de Charge (PRC'96)* (Presqu'île de Giens, Jul 1996).

[123] MIGUET, S., AND NICOD, J. A load-balanced parallel implementation of the Marching-Cubes algorithm. In *High Performance Computing Symposium'95* (Montréal, Juillet 1995), CRIM, pp. 229–239.

[124] MONTEIL, T. *Étude de nouvelles approches pour les communications, l'observation et le placement de tâches dans l'environnement de programmation parallèle LANDA.* PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, Nov 1996.

[125] MOREL, E., BRIAT, J., AND CHASSIN DE KERGOMMEAUX, J. PloSys: parallélisme OU et effets de bord sur système parallèle sans mémoire commune. In *Journées Francophones de Programmation Logique et programmation par Contraintes* (Clermont-Ferrand, 1996), pp. 131–145.

[126] MUELLER, F. Implementing POSIX Threads under UNIX: Description of Work in Progress. In *Proceedings of the 2nd Software Engineering Research Forum* (Melbourne Florida, Nov 1992), pp. 253–261.

[127] MUELLER, F. A Library Implementation of POSIX Threads under UNIX. In *Proceedings of the USENIX Conference* (1993), pp. 29–41.

[128] MUELLER, F., GIERING, E., AND BAKER, T. Implementing ada 9x features using posix threads: Design issues. *TRI-Ada* (September 1993).

[129] MULLENDER, S., VAN ROSSUM, G., TANENBAUM, A., VAN RENESSE, R., AND VAN STAVEREN, J. Amoeba – A Distributed Operating System for the 1990s. *IEEE Computer 23* (May 1990), 44–53.

[130] MUSSI, P., AND SIEGEL, G. The PROSIT sequential simulator: a test-bed for object-oriented discrete event simulation. In *7th European Simulation Symposium (ESS'95)* (Oct 1995).

[131] NAMYST, R., AND MEHAUT, J. PM^2: Parallel Multithreaded Machine. a computing environment for distributed architectures. In *ParCo'95 (PARallel COmputing)* (Sep 1995), Elsevier Science Publishers, pp. 279–285.

[132] NAMYST, R., AND MÉHAUT, J.-F. PM2: Parallel Multithreaded Machine. A multithreaded environment on top of PVM. In *2nd Euro PVM UG Meeting* (Lyon, Sept. 1995), pp. 179–184.

[133] NAMYST, R., AND MÉHAUT, J. MARCEL *: Une bibliothèque de processus légers.* Laboratoire d'Informatique Fondamentale de Lille, Lille, 1995.

[134] NAMYST, R., AND MÉHAUT, J. *PM^2 Parallel Multithreaded Machine: Guide d'utilisation.* Laboratoire d'Informatique Fondamentale de Lille, Lille, 1995.

[135] NAMYST, R., ROOS, J.-F., COURTRAI, L., AND MÉHAUT, J.-F. Implementation of parallel object languages using pvm. In *First PVM Users' Group Meeting* (Knoxville Tennessee, May 1993).

[136] OBJECT MANAGMENT GROUP. *IDL C++ Language Mapping Specification Version 94-9.* Framingham, MA, USA, Sep 1994.

[137] OBJECT MANAGMENT GROUP. *The Common Object Request Broker: Architecture and Specification version 2.0.* Framingham, MA, USA, Jul 1995.

[138] PEREZ, C. Load balancing HPF programs by migrating virtual processors. Rapport de Recherche 96-33, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Oct 1996.

[139] PEREZ, C. Utilisation de processus légers pour l'exécution de programmes à parallélisme de données: étude expérimentale. Rapport de Recherche 96-09, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Apr 1996.

[140] PLATEAU, B., AND ALL. Présentation d'APACHE. Tech. Rep. Apache TR-1, IMAG Institute, Grenoble, Oct. 1986.

[141] POWEL, M., KLEINMAN, S., BARTON, S., SHAH, D., AND WEEKS, M. SunOs 5.0 Multithreaded Architecture. In *Proceedings of the Winter 1991 USENIX Conference* (1991), pp. 65–79.

[142] POWELL, M., AND MILLER, B. Process migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* (1983), pp. 110–119.

[143] PROVENZANO, C. *Pthreads: an implementation of POSIX 1003.1c.* Available at http://www.mit.edu:8001/people/proven/pthreads.

[144] RIFFLET, J. *La Communication Sous Unix. Applications Réparties.* Ediscience International, 1994.

[145] ROMAN, J. Partitionnement algorithmique des données pour la factorisation de Cholesky par bloc de grands systèmes linéaires creux sur les calculateurs MIMD. *Lettre du Transputer et des Calculateurs Parallèles 24* (1994), 115–120.

[146] ROOS, J., COUTRAI, L., GEIB, J., AND MÉHAUT, J. Execution Replay of Parallel Programs. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing* (Jan. 1993), IEEE, pp. 429–434.

[147] ROOS, J.-F. *Mise au point d' applications distribuées pour environnement de développement basé sur une technologie objet.* PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Feb. 1994.

[148] ROOS, J.-F., COURTRAI, L., GEIB, J.-M., AND MÉHAUT, J.-F. Les Composants Actifs de Communication: Manuel de Programmation (V 2.0). Tech. Rep. ERA-115, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Sept. 1992.

[149] ROUCAIROL, C. STRATAGÈME : Une méthodologie de programmation parallèle pour les problèmes non structurés. Rapport de Recherche, PRiSM, Versailles, Dec 1995.

[150] SHAPIRO, M., GOURHANT, Y., HABERT, S., MOSSERI, L., RUFFIN, M., AND VALOT, C. SOS: An Object-Oriented Operating System - assesment and perspectives. *ACM Computing Systems 2*, 7 (1989), 287–337.

[151] STEIN, D., AND SHAH, D. Implementing Lightweight Threads. In *Proceedings of the Summer 1992 USENIX Conference* (1992), pp. 1–9.

[152] STELLNER, G., AND PRUYNE, J. Resource managment and checkpointing for pvm. In *EuroPVM'95 proceedings* (ENS Lyon, Sep 1995), LIP, Hermès, pp. 131–136.

[153] STROUSTRUP, B. *The C++ programming Language.* InterEditions, 1993.

[154] SUBRENAT, G. *Implémentations parallèles d'algorithmes de radiosité.* PhD thesis, Labri Université Bordeaux I, 1995.

[155] SUN MICROSYSTEMS. XDR: External Data Representation standard. Tech. Rep. RFC 1014, Sun Microsystems Inc., 1987.

[156] SUN MICROSYSTEMS. *Systems Services Overview Chapter 6: Lightweight Processes part number 800-1753*, 1988.

[157] SUNSOFT. *SunOs5.2 Guide to Multi-Thread Programming*, 1993.

[158] TANENBAUM, A. *Modern Operating Systems.* Prentice Hall International Editions, 1992.

[159] THACKER, C., STEWART, L., AND SATTERTHWAITE, E. Firefly: A Multiprocessor Workstation. *IEEE Trans. Comput. 37 8* (Aug. 1988), 909–920.

[160] U.S. DEPARTMENT OF DEFENSE. The ada language reference manual. Tech. rep., Department of Computer Science, Carnegie Mellon, USA, 1983.

[161] VON EICKEN, T., CULLER, D., GOLDSTEIN, S., AND SCHAUSER, K. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Int'l Symposium on Computer Architecture* (May 1992).

[162] WALLACH, D., HSIEH, W., JOHNSON, K., KAASHOEK, M., AND WEIHL, W. Optimistic active messages: A mechanism for scheduling communication with computation. Tech. rep., MIT Laboratory for Computer Science, 1995.