50376
1994
177

Numéro d'ordre : **1339**

USTL

Année : **1994**

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

Fred Hémery

Etude de la répartition dynamique d'activités sur architectures décentralisées

Thèse soutenue le 29 Juin 1994, devant la commission d'examen :

J.-P. Arcangeli	Maître de Conférences	Université Paul Sabatier, IRIT (*rapporteur*)
J.-M. Geib	Professeur	Université de Lille I, LIFL
J.-F. Méhaut	Maître de Conférences	Université de Lille I, LIFL
M. Mériaux	Professeur	Université de Lille I, LIFL (*président*)
J.-L. Roch	Maître de Conférences	IMAG-LMC (*rapporteur*)
P. Sallé	Professeur	Université Paul Sabatier, IRIT (*rapporteur*)
E.-G. Talbi		INPG-LGI
D. Trystram	Prof	IMAG-LMC (*rapporteur*)

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 20.43.47.24 Fax. 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. 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

Remerciements

Je tiens à remercier les membres du Jury:

Monsieur Michel Mériaux pour m'avoir fait l'honneur de présider ce jury, mais aussi pour la confiance qu'il m'a accordée en temps que directeur du LIFL, en m'autorisant à investir une part importante de mes heures de travail à ces travaux de recherche,

Messieurs Denis Trystram et Jean-Louis Roch pour avoir corapporté ce travail, leurs critiques et leurs remarques ont permis d'améliorer la rédaction du document,

Messieurs Patrick Sallé et Jean-Paul Arcangeli pour avoir accepté de corapporter cette thèse, et pour l'intérêt qu'ils ont manifesté pour ce travail,

Monsieur El-Ghazali Talbi pour avoir accepté d'être membre du jury,

Monsieur Jean-Marc Geib qui a dirigé ces travaux, ses remarques pertinentes ainsi que nos nombreuses discussions ont contribué à faire aboutir ce travail,

Monsieur Jean-François Méhaut membre de l'équipe PVC/BOX, nos discussions parfois houleuses m'ont permis de clarifier mes suggestions.

Je tiens également à remercier l'ensemble de l'équipe PVC/BOX pour l'ambiance de travail et le débat d'idées toujours très enrichissant, un merci tout particulier à Chrystel Grenot qui a relu le document.

Je joins à ces remerciements l'ensemble des membres de l'équipe technique, monsieur Henri Glanc pour le sérieux apporté dans le travail de reprographie de ce document et plus spécialement mes collègues de bureau Gilles Carin et Bernard Szelag qui ont bien voulu prendre en charge la plupart de mes tâches d'administration et de gestion du réseau informatique du LIFL pour me permettre de mener à bien ce travail.

Enfin je remercie les membres du LIFL qui contribuent à entretenir au sein du laboratoire une ambiance chaleureuse et passionnée.

Ce travail de thèse n'aurait pu être conduit à bien sans le soutien et les encouragements de ma femme Annie et de mes enfants Simon et Maxime qui m'ont permis d'oublier les tracas quotidiens, je leur dois beaucoup.

Table des matières

Introduction

Les machines parallèles sont devenues l'outil indispensable pour répondre aux grands challenges que se sont proposées de résoudre les sciences modernes dans les prochaines années (la modélisation du climat, l'étude des couches atmosphériques ou le génome humain sont inscrits dans le programme américain HPCC "*Highly Parallel Computing and Communication*" ou encore européen HPC) [CS93]. Dans le terme générique de "*machine parallèle*" on regroupe les machines massivement parallèles comme les machines MasPar[Bla90], la CM5[Thi91], ou encore la Paragon[Rou93], et les réseaux hétérogènes de stations de travail.

Une bonne utilisation de ces machines par une application nécessite un découpage de la charge qu'elle va produire au cours de son exécution, de façon à pouvoir ensuite la distribuer sur chaque noeud [1] de l'architecture. La charge se définit en fonction de la nature de l'application. Certaines applications s'organisent autour d'un traitement systématique à effectuer sur un volume de données important, ces applications développent un parallélisme sur les données, la charge étant quantifiée par le volume des données à traiter. D'autres applications manipulent des structures de données complexes et irrégulières qui demandent un traitement plus lourd, elles développent un parallélisme de traitement, la charge se mesure alors par le coût des traitements. Enfin certaines applications ont une charge qui évolue dynamiquement au cours de l'exécution, le volume et la structure des données à traiter sont transformés ou bien les traitements à effectuer sont modifiés en fonction des données. Dans ce cas le parallélisme du traitement doit s'adapter aux données ou inversement le parallélisme sur les données doit s'adapter aux traitements suivant l'architecture de la machine.

1. La composition d'un noeud varie d'une architecture à l'autre, mais dispose au minimum d'une unité de traitement

Le travail qui consiste à distribuer la charge que représente l'exécution de ces applications sur des architectures de machine parallèle est largement abordé comme thème de recherche. De nombreuses taxinomies des différents modèles de distribution ont déjà été données [CK88, BSS91, CT93]. Elles développent des classes de modèles qui s'appliquent à une période particulière dans le développement de l'application (par exemple les modèles de placements statiques qui utilisent les résultats de l'analyse du source d'une application). Une autre classification nous parait interessante car basée sur deux critères permettant de mieux identifier un modèle de distribution. Un premier critère précise l'environnement dans lequel va s'exécuter la distribution de la charge, cet environnement étant caractérisé par l'architecture de la machine et le type de charge produite par l'application, le deuxième critère précise le moment dans le cycle de développement de l'application pendant lequel va être traitée la distribution de la charge. Détaillons ces deux critères.

Environnement de distribution

Derrière cette notion, on retrouve essentiellement l'architecture de la machine. Les machines dites "*à parallélisme de données*", utilisent généralement un mécanisme d'exécution SIMD qui aborde la distribution de la charge en faisant une répartition la plus appropriée possible des données. Ce type d'architecture est bien adapté à un traitement systématique et récurrent des données, mais souffre d'un manque d'adaptation aux structures de données irrégulières. Les autres machines dites "*à parallélisme de traitement*" ont un mécanisme d'exécution SPMD (P pour Program) ou MIMD et ajoutent nombre de problèmes, notamment la synchronisation des processus concurrents. Les exécutions concurrentes entraînent en effet des accès simultanés à des ressources critiques dont il faut assurer la cohérence. La distribution de la charge doit résoudre cette fois une répartition des traitements adaptée à la répartition des données (ou inversement). Certains modèles de programmation (avec les langages parallèles orientés objets par exemple) peuvent conduire à un placement conjoint des données et des traitements dirigé par le programmeur. A cela il faut ajouter les systèmes d'exploitation répartis qui sont de plus en plus étudiés comme support de distribution d'une application [Fol93].

Les caractéristiques de l'application font aussi partie de l'environnement de l'outil de distribution. On peut distinguer là encore plusieurs spécificités :

- La granularité des tâches qui composent l'application à répartir;

  Une tâche représentant la suite d'une dizaine d'instructions ne peut être répartie de la même manière qu'une tâche réalisant une compilation.

- Le volume des données à traiter; La répartition d'une base de données et des

traitements qui l'accompagnent demande qu'un soin particulier soit donné en premier lieu à la répartition des fichiers et des structures de données qui la composent [Nic91], la répartition du traitement étant largement dépendante de la localisation des données. Certaines applications ont au contraire, peu de données de base, mais ou bien elles utilisent des traitements complexes, ou alors, elles créent dynamiquement une structure de données irrégulière qu'il faut répartir pour favoriser un traitement parallèle.

- La complexité des traitements.

Enfin les algorithmes de répartition se distinguent par leur objectif. Si leur utilisation la plus courante consiste à réduire le temps de réponse d'une application, cet objectif peut être complété par des contraintes temporelles plus strictes dans le cadre d'applications temps réels ou des contraintes de tolérances aux pannes qui modifient le comportement du répartiteur de charge.

Moment d'intervention de la distribution

La deuxième manière de classifier les modèles de répartition est de déterminer le moment où sont abordées les directives de distribution dans le développement d'une application. On divise en quatre moments :

- A la création de l'application, ce qui permet d'identifier les composants qu'il va falloir distribuer.

- Lors de l'analyse du source, afin de déterminer la complexité des composants ainsi que (lorsque c'est possible) les relations qui existent entre eux (précédence, communication, ...), et d'aboutir à une distribution statique.

- Lors du chargement de l'application, pour faire un placement initial des composants qui reprend les informations obtenues dans les périodes précédentes pour les adapter à l'architecture cible et à la charge courante de la machine parallèle.

- Enfin au cours de l'exécution, pour traiter le problème du placement des composants lorsqu'il y a des créations dynamiques et la répartition de la charge lorsque le besoin s'en fait sentir, par exemple lorsqu'une modification de l'architecture survient.

Cette classification va nous permettre d'identifier notre travail et le contexte dans lequel il se situe.

Problématique

Le projet PVC/BOX du LIFL a pour ambition de définir un modèle d'exécution pour langages parallèles orientés objets en définissant une architecture logicielle au dessus des systèmes d'exploitations des machines parallèles du type multi-ordinateurs. Cette approche se justifie d'une part par les bonnes qualités (génie logiciel, modélisation du monde réel) reconnues pour la programmation basée sur l'approche objet [Weg90], et d'autre part par l'émergence de langages à objets-actifs [YT87] qui permettent d'exploiter les architectures parallèles. La thèse de Luc Courtrai[Cou92] introduit un grain unique de distribution des activités et des données: Le Composant Actif de Communication (CAC) qui peut être assimilé à un acteur simplifié. Le projet repose d'abord sur la conception d'un run-time utilisant les Cacs et conduisant à un environnement de programmation parallèle intégrant le déverminage des programmes parallèles [Roo94], un ramasse-miettes [Dum92] et un outil de distribution dynamique. C'est la composante PVC du projet. La deuxième composante, suite à la mise en place d'une spécification, consiste en la réalisation d'un langage à objets actifs: le langage BOX[GGG93] adapté à PVC. BOX est un langage orienté objet fortement typé qui permet de manipuler des entités passives (Objets) et actives (Acteurs) communicant par messages.

Le modèle d'exécution dans PVC développe un parallélisme important de processus d'une granularité fine. Ces processus communiquent par envoi de messages asynchrones. L'exécution d'une application est lancée par un processus particulier représentant l'objet principal de l'application. Cet objet va lancer des requêtes pour l'exécution de méthodes qui vont se traduire par des demandes de création d'autres processus. L'adresse d'un processus peut être transmise dans un message, ce qui induit que le graphe des communications entre les processus évolue dynamiquement.

Ma participation au projet m'a conduit à m'intéresser, dans un premier temps, plus particulièrement au problème de la distribution des CACs sur les machines où une version du run time PVC a été implantée:

- Une machine à base de transputers: La **Multiculster-II** de chez *Parsytec* composée de 32 noeuds qui sont entièrement reconfigurables au dessus du système d'exploitation *Helios*.
- Un **réseau de stations de travail** *SUN* au dessus du système d'exploitation *SunOs*.

Dans un deuxième temps, suite à la réalisation d'un outil de distribution dynamique de charge, nous avons développé une plate-forme d'évaluation de la distribution de la charge permettant de déterminer le bon paramétrage de l'outil de distribution réel et cela pour une application et une architecture données.

La distribution de charge dans le projet PVC

En tenant compte des caractéristiques du projet PVC et pour satisfaire l'objectif d'une distribution des cacs qui conduise à réduire le temps de réponse de l'application, nous devons nous poser les questions suivantes :

- Quand faire la répartition ?

  Des éléments de réponse à cette question sont donnés dans la thèse de L. Courtrai qui propose la structure de module pour distribuer statiquement le code des processus. Seuls les noeuds avec un module contenant le code du processus pourront l'exécuter. Le module est donc une première étape de distribution, qui peut conduire à dupliquer l'ensemble du code sur l'ensemble des noeuds. Cependant certaines architectures de machines ou certaines applications peuvent s'éloigner de cette solution pour des raisons d'hétérogénéité par exemple.

  La distribution intervient aussi au moment du chargement des modules dans les noeuds, et pendant l'exécution de l'application pour donner une régulation qui réponde au nombre important de demandes de créations dynamiques de processus et à l'évolution de leurs relations.

- Quelles sont les stratégies de répartition les plus adaptées ?

  Le choix d'une stratégie de répartition nous l'avons vu dépend de l'environnement dans lequel elle va se trouver. Il est clair que dans le modèle d'exécution PVC un placement statique ne suffit pas à garantir une bonne répartition de la charge au cours de l'exécution. De nombreux modèles de répartition dynamique de charge existent [CK88, BSS91], ils reposent sur une connaissance plus ou moins globale (*Politique d'information*) d'une représentation (*Indicateur de charge*) de l'état de la charge courante de la machine. Les périodes où la répartition intervient au cours d'une exécution (*Politique de transfert)*, et le choix du noeud (*Politique de localisation*) vers lequel on va diriger l'élément de charge à distribuer nécessitent de plus une connaissance de l'application. La notion de "placement multi critères" [Fol93], pour répartir la charge engendrée par des applications à granularité importante, est ici reprise en l'adaptant à une granularité fine.

Pour faire un choix de répartition qui soit fonction des caractéristiques de l'application, il nous fallait développer un outil qui nous permette d'analyser son comportement. Nous avons choisi de créer un langage (GENESE) permettant d'exprimer à partir d'un nombre réduit d'opérateurs le comportement d'applications. Celui-ci est en fait un générateur d'événements permettant de tracer une exécution.

Nous avons développé une plate-forme d'aide à la paramétrisation d'une stratégie de placement dynamique qui soit fonction de l'architecture de la machine cible et de l'application étudiée. Cette plate-forme accepte trois types d'entrées :

- Une description de l'application constituée de processus communicants à l'aide du langage GENESE qui permet d'exprimer la génération d'une suite d'événements caractéristiques facilitant l'analyse de son exécution.
- Une description de l'architecture permettant de spécifier pour chaque noeud ses caractéristiques qualitatives (puissance relative, taille mémoire, ressources particulières).
- Une combinaison de paramètres pour les stratégies de distribution des processus communicants.

Notre modèle de distribution a la volonté de rechercher une solution qui prenne en compte les caractéristiques de l'application grâce à l'exploitation des résultats obtenus sur la plate-forme d'évaluation, tout en prenant en compte l'état courant de la machine au moment de l'exécution. Par rapport à la classification introduite plus haut, notre environnement de distribution se situe dans un contexte qui utilise un parallélisme de traitement pour des applications composées de tâches d'une granularité fine et un taux de création dynamique d'activités important. Notre intervention dans la distribution se situe à deux moments :

- Au cours du regroupement et du chargement du code des activités sur les noeuds de l'architecture;
- Pendant l'exécution.

Notre travail de distribution est basé sur l'étude du comportement de l'application obtenue grâce à notre plate-forme d'évaluation.

Plan de la thèse

Dans le chapitre 1, après avoir précisé les critères importants qui constituent l'environnement d'un algorithme de placement, nous présentons les informations récoltées tout au long du développement d'une application jusqu'à son exécution pour aider à une utilisation optimale de la machine parallèle. Nous donnons ensuite un état de l'art sur les différents modèles d'algorithmes de placement généralement utilisés : les algorithmes de placement statique, et les algorithmes de placement dynamique.

Nous revenons sur le contexte de notre travail dans le chapitre 2. Le lien du projet PVC avec la programmation à objets actifs introduit des particularités

quant aux méthodes à utiliser pour distribuer la charge sur une architecture décentralisée. Deux niveaux de distribution sont utilisés : 1) La distribution du code des activités dans les modules; 2) Le placement dynamique des activités.

Le chapitre 3 fait une présentation de la plate-forme pour l'aide à l'étude comportementale d'une application grâce au langage GENESE et à l'adaptation des paramètres caractéristiques d'une stratégie de répartition: 1) Indicateur de charge; 2) Politique d'information; 3) Politique de transfert; 4) Politique de localisation.

Le chapitre 4 donne un aperçu des possibilités offertes par la plate-forme. Après avoir proposé une méthode de détermination des paramètres pour la modélisation de l'architecture, et donné une présentation des outils de mesure qui seront utilisés pour faire une analyse du comportement des applications, un certain nombre d'études de cas caractéristiques sont étudiés. Une synthèse sur l'aide à la détermination d'une stratégie de placement dynamique adaptée à la fois à l'application et à l'architecture est alors proposée.

En annexe sont donnés la syntaxe du langage GENESE, suivie du code des applications présentées dans le rapport.

Chapitre 1

Etat de l'art sur les problèmes de placement

1.1 Introduction

Depuis une vingtaine d'années, l'informatique n'est plus seulement utilisée pour effectuer des travaux de gestion, mais permet aussi de traiter des problèmes de plus en plus complexes comme la modélisation des prévisions météorologiques, la modélisation de la turbulence créée par les ailes d'un avion en vol, la recherche et le stockage du génome humain, le traitement et la synthèse d'images, etc ....

Cette modification de la complexité des applications demande aux calculateurs une adaptation en puissance et en rapidité. C'est ainsi que l'on a vu apparaître sur le marché des machines affichant des capacités de traitement de plus en plus importantes(IBM 30-90). Mais une fois les limites technologiques atteintes, il a fallu faire évoluer le modèle d'exécution séquentiel vers un modèle d'exécution parallèle (Cray 1) et ensuite décentralisé (Intel IPSC/2). Aujourd'hui il existe un large catalogue de machines à architecture parallèle qui aide à repousser les limites du "concrètement calculable" en terme de volume d'informations traités. Paradoxalement, actuellement le problème n'est plus de disposer d'une puissance de calcul suffisante pour traiter un problème complexe, mais plutôt de trouver des méthodes et des outils permettant d'utiliser de façon optimale la puissance des machines.

Un des points essentiels pour bien utiliser une machine à architecture parallèle est de trouver un bon placement des composants d'une application sur les noeuds

de la machine. Quand la taille de l'application, le nombre de ses composants, le volume des données et le nombre de noeuds de la machine parallèle atteignent des valeurs importantes, il devient impossible pour l'utilisateur de faire un placement intuitif efficace des entités constituant son application. La mise en place d'outils permettant de faire un placement plus ou moins automatique devient alors indispensable pour l'aider dans cette tâche.

Nous allons nous attacher dans un premier temps à définir les caractéristiques importantes de chacun des acteurs intervenant dans le problème du placement d'une application sur une machine parallèle.

- l'application;

  Pour que l'on puisse la placer, une application doit avoir été découpée en entités élémentaires. Ce sont ces entités qu'il faut analyser pour pouvoir définir les caractéristiques du placement. On peut identifier les entités passives,ce sont les données traitées, et les entités actives qui vont effectuer des traitements sur les entités passives. Depuis l'introduction de la programmation par objets on trouve maintenant des entités qui regroupent à la fois les données et les traitements sur ces données; ce qui introduit une nouvelle problématique de placement.

- la machine parallèle

  Le problème du placement ne peut pas être abordé de la même façon sur une machine parallèle disposant d'un modèle d'exécution SIMD que sur une machine avec un modèle d'exécution MIMD. Dans le premier cas seul un placement des données peut avoir un sens, étant donné que le code est séquencé par un contrôleur unique. Dans le second cas on peut choisir de placer les données et ensuite les traitements en fonction des données ou l'inverse, ou encore les deux en même temps dans une approche objet.

Après une présentation des différents types de composants qui peuvent être rencontrés, et les différentes architectures de machine parallèle, nous donnerons la liste des objectifs que doit atteindre un algorithme de placement. Nous aborderons le problème du placement en identifiant les différentes étapes du développement d'un logiciel. Pour chacune des étapes nous allons déterminer les informations qui peuvent faciliter son placement, et définir les intéractions qui existent entre elles. Nous terminerons par une présentation des deux modèles de stratégies de placement très largement développés:

- le placement statique;

- le placement dynamique.

1.2 Identification des entités à placer

Avant de définir une stratégie de placement, il faut identifier l'entité qui va être placée. Dans ce paragraphe nous allons donner les différents types d'entités qu'un algorithme de placement peut être amené à placer. Les caractéristiques de l'entité désignée doivent permettre d'atteindre facilement le but de l'algorithme de placement (voir section 1.4). Nous allons présenter pour chaque entité les caractéristiques qui vont être utilisées. Enfin pour chacune des entités nous allons indiquer le moment où l'algorithme de placement est déclenché.

1.2.1 Placement de fichiers

Certaines applications manipulent un gros volume d'informations. Les applications bancaires et dans un cadre plus général les applications utilisant des bases de données réparties doivent pouvoir garantir des temps de transactions minimaux et imposent des contraintes de sécurité souvent très importantes. L'entité qu'il convient de bien placer dans ce cas est le **fichier** [BCS89] contenant un ensemble d'informations structurées. Un fichier est indivisible, les critères qui le caractérisent sont la taille mémoire nécessaire à son stockage, le volume moyen des requêtes qui peuvent être des modifications ou des interrogations, et cela pour chacun des sites d'où proviennent les requêtes. Un fichier peut exister en plusieurs exemplaires répartis sur les noeuds de l'architecture, ceci afin d'augmenter la probabilité d'avoir un client proche lors d'une transaction et de permettre des consultations en parallèle. En contre partie, plus le nombre de copies est important, plus la gestion de la cohérence des informations sera coûteuse. L'objectif qui consiste à obtenir un coût minimal d'exécution se traduit par la détermination d'un nombre de copies optimal pour chacun des fichiers et par le placement optimal de ces copies sur les noeuds de l'architecture. En général le problème est abordé fichier par fichier, en considérant qu'aucune relation ne les relie. Dans une seconde phase on ajoute les contraintes permettant d'intégrer dans le problème du placement trois types d'objectifs adressés à l'ensemble des fichiers : un niveau de disponibilité, le délai maximum admis d'une transaction, les limites de stockage des noeuds. Le placement des fichiers peut être fait avant le lancement de l'application, une redistribution pouvant se faire par migration au cours de l'exécution, dans le cas d'une modification des conditions d'utilisation. Le problème du placement de fichier peut aussi être utilisé pour résoudre le problème du placement du code des processus pendant la phase de chargement de l'application sur les noeuds de la machine parallèle. Chaque code d'un processus correspond alors à un fichier du modèle présenté, qu'il convient de placer au mieux pour permettre une exécution ayant un coût minimal pour l'application [Lit92].

1.2.2 Placement de données

Les phénomènes ou les systèmes complexes dont on veut faire une modélisation afin d'en étudier le comportement par simulation sur ordinateur nécessitent l'utilisation de **structures de données** sur lesquelles on applique des traitements. Le placement des données est utilisé comme un moyen de distribution de la charge notamment dans le cas de l'utilisation des architectures SIMD (voir 1.3.1). L'exemple que l'on peut donner est celui d'une matrice dans le calcul numérique, ou encore une forêt d'arbre n-aire de rayons de lumière dans les techniques de lancer de rayons ou les données d'une scène en synthèse d'images.

Plus récemment, on a vu émerger de nombreuses méthodes de résolution de problèmes hérités de l'intelligence artificielle ou de la recherche opérationnelle, dont le traitement consiste à parcourir une structure de données qui évolue dynamiquement en fonction des résultats aux traitements précédents. Par exemple l'algorithme A^* fait le parcours d'un arbre qui se construit dynamiquement en fonction du traitement effectué. L'étude de la parallélisation du parcours d'un espace de solutions a été très souvent abordée, dans [PFK93, KK92, KR91, Rou93] on trouvera une description détaillée des différentes techniques. La méthode la plus utilisée consiste à répartir des sous arbres de l'espace de recherche des solutions sur les noeuds de la machine. Le traitement d'un sous arbre peut conduire à arrêter son évaluation, tandis qu'il peut créer d'autres sous-arbres ailleurs. Ce type de fonctionnement demande qu'une redistribution dynamique des données (les sous arbres) soit faite pour rééquilibrer la charge des traitements.

Pour les machines SIMD il est important de bien répartir les données pour limiter les problèmes d'accès concurrents à la mémoire et éviter que des processeurs de la machine soient inactifs. Le placement peut être systématique quand on ne s'intéresse qu'à la taille du problème, par exemple la distribution des données d'un vecteur dans une mémoire organisée en bancs, cela consiste alors à résoudre un problème de recouvrement entre la structure de donnée et l'architecture disponible. Certaines études [Law75, IC82] vont plus loin en placant la structure de donnée en fonction des valeurs qu'elle contient comme par exemple le traitement des matrices creuses qui conduisent à un traitement particulier de l'application. Le placement d'une structure de donnée se fait avant l'exécution de l'application, au moment du chargement, et aussi pendant l'exécution quand la structure de donnée évolue dynamiquement et de façon imprévisible statiquement.

1.2.3 Placement de tâches et de processus

Dans la plupart des cas, une application complexe est composée de **tâches** indépendantes qui s'échangent les données dont elles ont besoin pour fonctionner. Les tâches sont elles mêmes composées d'une collection d'entités qui représentent

l'activité de la tâche: les **processus**. Le déroulement de l'exécution d'une application peut être représenté par un graphe de dépendance des tâches. L'entité qu'il convient de placer pour une utilisation efficace de la machine parallèle est la tâche et les processus qui la composent. Le placement de tâches et de processus est très largement abordé, on pourra trouver une bonne synthèse dans [AP88, BSS91, MT91]. On distingue le placement de tâches du placement de processus par le type de l'application, la granularité des composants, et l'architecture de la machine.

1.2.3.1 Placement de tâches

Pour le placement de tâches les critères retenus sont: l'occupation mémoire prise par le code et des données. Dans le cas d'une architecture hétérogène (réseau de stations de travail, voir 1.3.1.3) il faut retenir le type de code, les besoins spécifiques de ressources particulières [TH86] (disque, CoProcesseur, Carte graphique). Le placement des tâches peut aussi être une façon de répondre au problème de la tolérance aux pannes [BAAD91], en dupliquant le code en plusieurs exemplaires placés sur des sites différents. Le placement de tâches se fait suite à l'analyse de l'application et au chargement, il doit vérifier les contraintes définies plus haut.

1.2.3.2 Placement de processus

Dans le placement de processus on s'intéresse davantage à l'activité que représente la tâche que l'on cherche à placer. Dans ce cas les critères à retenir pour le placement sont le coût ou le temps d'exécution (suivant le but du placement), et le volume de communication entre les processus. Les deux critères peuvent être indissociables. Lorsque les phases de calcul sont entrelacées avec des phases d'attentes de communication. Dans cd cas les deux critères pris indépendemment sont inadaptés pour mettre en place une stratégie de placement. Il est aussi très important de connaître le comportement global de l'exécution d'une application, c'est à dire savoir si l'activité des processus crée dynamiquement ou non de nouveaux processus. Le placement des processus est fait suite à l'analyse de l'application, au moment du chargement pour adapter le placement statique à l'architecture cible, et peut aussi intervenir au cours de l'exécution quand les modifications du comportement de l'application dans son ensemble sont importantes.

1.2.4 Le placement d'objets

Le paradigme de programmation objet a introduit de nouveaux types d'entités à placer. Les acteurs [Agh86] sont la réunion dans une même entité d'un *script* qui représente le code de traitement écrit par le programmeur et des *ac-*

quaintances qui représentent les données de l'objet. Les acteurs communiquent par envoi de messages asynchrones. Un peu plus tard et pour des raisons d'efficacité est introduit la notion d'objet serveur [Ame87] qui contient un *body* (code réactif de l'objet) qui attend les messages de requêtes sur l'objet et exécute les méthodes (code déclenché par le body) sur les données de l'objet. Un objet est une instance (une copie) d'une classe (un moule) avec des données particulières. A chaque demande de création de l'instance d'une classe, il y a création d'un nouvel objet actif. Le placement des objets revient donc à déterminer le noeud sur lequel on va placer la nouvelle instance d'une classe. Les critères de placement d'un objet sont liés à la représentation qui en est faite sur la machine parallèle [Cou92], et au type d'objet concerné. Le code et son coût par voie de conséquence est dépendant de l'utilisateur qui l'a écrit. Il y a donc des objets avec un code très coûteux en temps de calcul et d'autres avec un code très léger. Il y a des objets qui ont un taux de création d'instances important. Les données d'un objet peuvent être de deux natures différentes, soit la donnée est une valeur simple et elle est stockée dans l'objet, soit elle représente une référence sur un autre objet, et dans ce cas pour y accéder l'objet qui posséde la référence va devoir envoyer un message vers l'objet actif pointé par la référence. Cette propriété entraine un taux de communication entre objets qui peut être important et qu'il est souhaitable de réduire par un placement adéquat.

La représentation des objets sur machine parallèle est variable et les possibilités de placement le sont tout autant. Nous reviendrons plus en détail sur la présentation du monde objet qui fait partie du contexte de notre étude.

1.2.5 Conclusion sur les entités

Dans ce paragraphe nous avons fait une présentation des entités et de leurs critères qui peuvent être manipulés par un algorithme de placement. On distingue deux grandes classes, dans l'une on regroupe les entités passives comme les fichiers, les structures de données plus ou moins complexes, et dans l'autre les entités actives comme les tâches et les processus. L'utilisation de l'une ou de l'autre des classes pour réaliser une bonne distribution de la charge sera fonction de l'architecture (voir le paragraphe 1.3), mais aussi de leur importance relative vis à vis de l'application. Enfin nous avons introduit la notion de répartition de charge par le placement des objets qui regroupent traitements et données. Dans les chapitres suivants nous nous intéresserons plus particulièrement à l'étude du placement d'objet actifs comme le moyen de répartir la charge induite par l'exécution d'une application utilisant le paradigme objet.

1.3 Impact de l'architecture de la machine cible sur le placement

L'algorithme de placement est fortement dépendant de la machine cible à laquelle il est destiné. Le placement doit prendre en compte le type de contrôle d'exécution de la machine SIMD (Single Instruction stream Multiple Data streams), SPMD (Single Program Multiple Data streams), MIMD (Multiple Instruction streams Multiple Data streams), le caractère homogène ou hétérogène des noeuds de la machine, la topologie du réseau d'interconnexion des noeuds. Aujourd'hui on considère aussi un réseau de stations de travail comme une machine parallèle, nous examinerons donc ce cas particulier.

Dans ce paragraphe, après quelques rappels sur les caractéristiques importantes de chacune de ces architectures, nous allons détailler comment elles interviennent dans la construction d'un algorithme de placement.

1.3.1 présentation des différentes architectures

1.3.1.1 les machines vectorielles pipelines et les machines tableaux

- Les machines vectorielles pipelines

  Les premières machines vectorielles pipelines étaient mono-processeur (Cray 1, Cyber 206), le parallélisme était introduit par l'utilisation d'unités fonctionnelles pipelinées. Ces machines sont dédiées à la manipulation de vecteurs grâce aux registres vectoriels et un jeu d'instructions spéciales. Aujourd'hui il existe plusieurs machines vectorielles pipelines avec plusieurs processeurs (Cray X-MP, Cray Y-MP), ces machines restent cependant SIMD, la machine Cray C-90 avec 16 processeurs, quant à elle, est une machine MIMD vectorielle pipeline (voir Fig. 1.1).

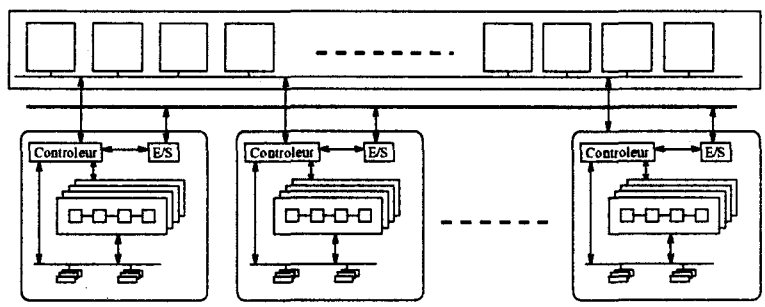


FIG. 1.1 - *Architecture d'une machine vectorielle pipelines multi-processeurs*

L'ensemble de ces machines travaillent sur un ensemble de registres scalaires ou vectoriels qui sont chargés (qui chargent) à partir d'une mémoire globale

à l'ensemble des processeurs. Pour permettre un accès efficace, elle est découpée en bancs qui sont multi-accès. L'accès à la mémoire reste néanmoins un goulot d'étranglement important pour ce type de machine.

- Les machines tableaux

  Les machines tableaux ou encore machines massivement parallèles (MasPar, DEC mpp) se composent d'un grand nombre (jusqu'à quelques dizaines de milliers) de processeurs élémentaires (PE). Les PEs sont sous la responsabilité d'un séquenceur. La présence de plusieurs séquenceurs permet de diviser la machine. Chaque PE dispose d'une mémoire locale qui contient les données qu'il va devoir traiter; le processeur reçoit le microcode de l'instruction à exécuter par le séquenceur. Un réseau d'interconnexion permet la communication entre deux PE quelconques.

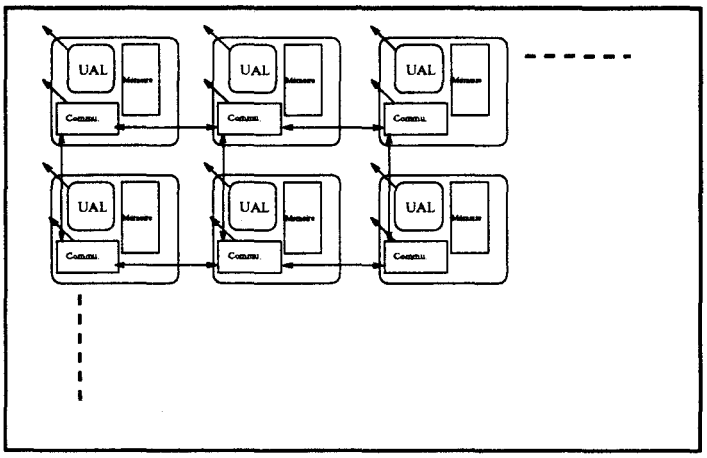


FIG. 1.2 - *Architecture d'une machine tableau*

Les machines vectorielles pipelines et les machines tableaux ont un contrôle d'exécution SIMD (sauf pour la dernière génération de machines, comme la Cray C-90, qui tendent vers un contrôle d'exécution SPMD ou MIMD), qui traitent le parallélisme de données. Ce type de parallélisme permet de prendre en charge efficacement des applications ayant un traitement systématique à effectuer sur des structures de données régulières comme les matrices. L'architecture est à fortiori homogène. Le réseau d'interconnexion des machines vectorielles pipelines permet uniquement de re-synchroniser les processeurs, la mémoire étant globale à l'ensemble des noeuds. Pour les machines tableaux il existe plusieurs niveaux de communication permettant de fournir des données soit aux voisins immédiats (4 ou 8), soit (suite à une reconfiguration dynamique du réseau globale de communication) à un noeud quelconque (voir Fig. 1.2).

1.3.1.2 les machines de type multiprocesseur et multicomputer

Contrairement aux machines SIMD, les machines de type multiprocesseur ou multicomputer, se caractérisent par un contrôle d'exécution complétement décentralisé, chaque noeud disposant d'un compteur ordinal. On retrouve là aussi des machines que l'on peut qualifier de massivement parallèle car elles peuvent disposer jusqu'à un millier de processeurs. On distingue deux types d'architectures suivant qu'elles ont une mémoire commune (**les multiprocesseurs**) ou une mémoire distribuée (**Les multicomputers**).

les multiprocesseurs

Les multiprocesseurs(voir Fig. 1.3) sont caractérisés par un **ensemble de processeurs** plus ou moins évolués pouvant accéder à une **mémoire globale** par l'intermédiaire d'un **réseau d'interconnexion**. La mémoire peut être organisée de deux façons différentes. Elle peut être organisée en bancs mémoire (voir Fig 1.3 (a)), chaque accès mémoire demande alors un temps constant quelque soit l'adresse accédée, et quelque soit le processeur qui fait la demande, ou chaque partie de la mémoire est la propriété d'un processeur qui peut autoriser l'accès par les autres processeurs (voir Fig 1.3 (b)). Dans ce cas l'accès à la mémoire locale sera moins coûteux que l'accès à une mémoire distante.

Le réseau d'interconnexion joue un rôle très important dans la capacité à traiter les demandes d'accès à la mémoire et éviter qu'elle ne devienne un goulot d'étranglement. Il influence directement le coût des communications. Dans [BYA89, Hsu93] on trouvera une étude des différents types de réseau d'interconnexion. Ce type d'architecture limite le nombre de processeurs pour des raisons techniques liées à la réalisation du réseau d'interconnexion, et par l'incidence direct sur le coût d'accès à la mémoire.

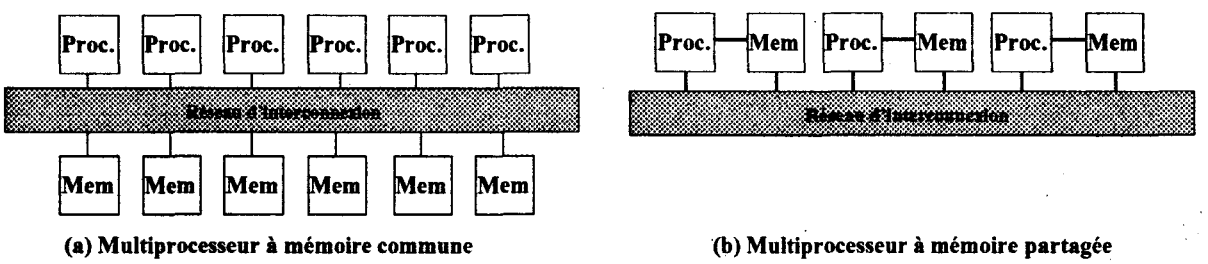


FIG. 1.3 - *Architecture d'une machine multiprocesseur*

les multicomputers

Contrairement aux multiprocesseurs, les multicomputers ne disposent pas d'une mémoire globale, chaque processeur dispose d'une mémoire qui lui est propre et inaccessible directement par les autres. Chaque noeud de la machine communique par messages avec les autres par l'intermédiaire d'un réseau de communication (voir Fig. 1.4). Pour accéder à une donnée présente sur un noeud distant, il faut envoyer un message sur le réseau. Le temps mis par le message pour arriver à destination est fonction du nombre de noeuds intermédiaires à traverser, du temps nécessaire pour parcourir un lien entre deux noeuds, et enfin de la taille du message. $T = T_pD + (L/B) * (D + 1)$, avec T_p temps de traitement par noeud traversé, D le nombre de noeuds traversés, L la longeur du message et B la bande passante du lien. Dans [AS88], on trouvera une comparaison entre l'architecture multiprocesseur et l'architecture multicomputer.

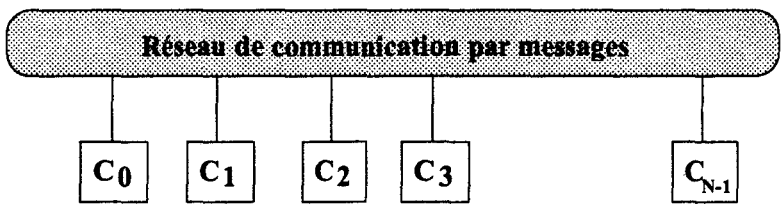


FIG. 1.4 - *Architecture d'une machine multi-ordinateurs*

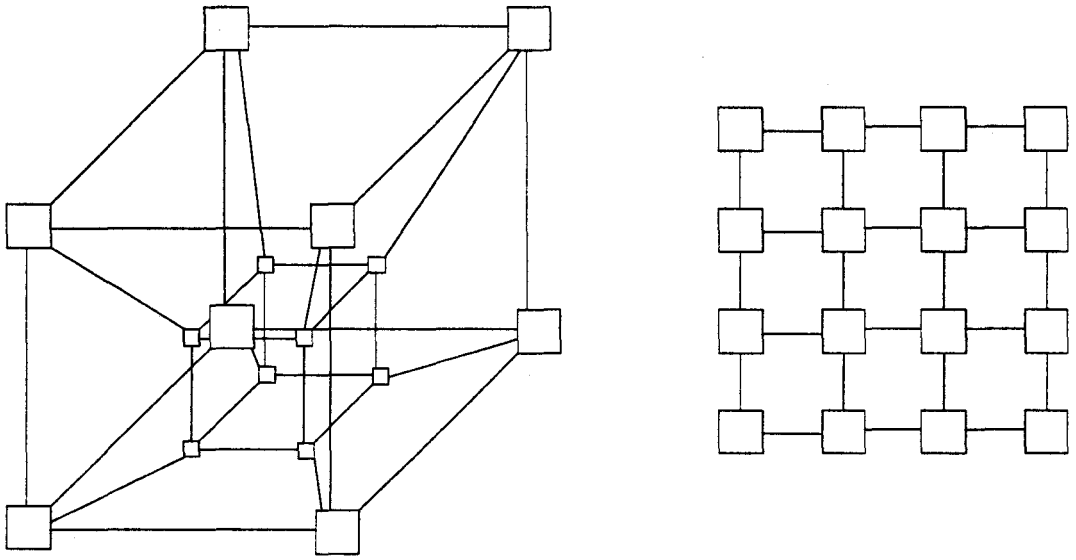
La topologie du réseau de communication constitue une donnée importante de l'architecture de la machine, elle est definie par

- N le Nombre de noeuds de la machine parallèle,
- la **distance** d_{ij}entre n_i et n_j, qui correspond au nombre de noeuds à traverser pour du noeud n_i atteindre le noeud n_j
- le **diamètre** D qui correspond à la valeur maximale de la distance $D = Max(d_{ij}, \forall i, j \leq N)$,
- le **degré** qui indique le nombre de voisins immédiats de chaque noeud

Par exemple une topologie représentant un hypercube de degré quatre, a un diamètre et un degré égal à quatre (voir Fig. 1.5(a)), une topologie représentant une grille 4×4 a un degré qui vaut quatre et un diamètre six (voir Fig. 1.5(b)).

1.3.1.3 les machines à noeuds distribués faiblement couplés

Etant donné que la puissance des stations de travail ne cesse d'augmenter, certains ont pensé qu'il pouvait être intéressant de considérer une telle station



(a) Hypercube de degré 4 (b) Grille 4 x 4

FIG. 1.5 - *Topologie du réseau d'une machine multicomputers*

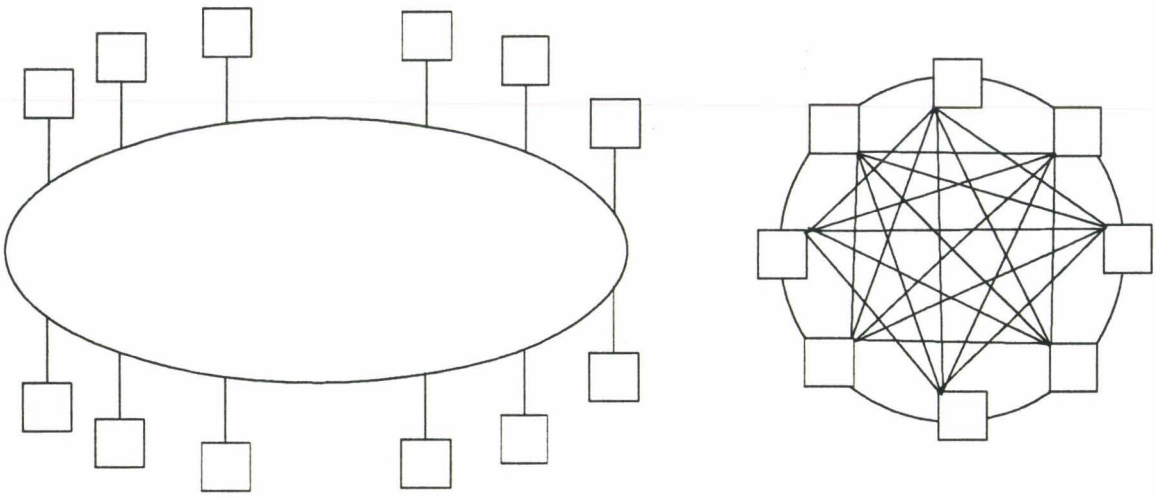
comme un noeud d'une machine parallèle, et le réseau local qui les relie comme le réseau de communication. On dispose alors d'une architecture, qui peut être hétérogène, dont la topologie a un degré égal au nombre de stations de travail présentes sur le réseau local et donc un diamètre égal à un (voir Fig. 1.6). Ce type d'architecture a fait émerger de nombreuses librairies de communication inter-processus que ceux-ci soient locaux à un site ou distribués sur le réseau. L'une des plus connues étant PVM [BDG+93].

1.3.2 Influence de l'architecture sur le placement

La présentation des différentes architectures de machine parallèle nous indique que l'algorithme de placement doit tenir compte des caractéristiques essentielles de chacune pour être conçu et utilisé efficacement.

les machines SIMD

Pour cette architecture, il s'agit de placer les données de l'application, et cela en fonction de la structuration de la mémoire de la machine. Un mauvais choix dans le placement des lignes d'une matrice dans les bancs mémoire d'une machine vectorielle pipeline peut aboutir à un accès séquentiel de chaque élément d'une ligne, si la ligne se trouve sur un même banc mémoire, et ainsi faire perdre tous les avantages de cette architecture dans la suite du traitement. Il en est de même pour le placement des données dans la mémoire locale des PEs d'une machine



(a) Réseau en anneau **(b) Réseau en étoile**

FIG. 1.6 - *réseau de stations de travail constituant un système distribué faiblement couplé*

tableau.

Le placement des données pour ce type d'architecture est fait par le programmeur qui indique dans le source de son application des directives (ALIGN, et autres astuces de déclaration des structures de données [Law75, IC82]) qui seront traduites par le compilateur. On ne peut pas, bien évidemment, parler de placement de tâches ni de placement de processus dans le cas d'une architecture SIMD, puisqu'il n'y a qu'un seul séquenceur qui exécute un seul programme. On peut cependant noter que les nouvelles générations de machines introduisent une dose de *multi-tasking* (pour la Cray C-90). Dans le cas d'un placement dynamique des données, le code de l'application doit être arrété pour mettre en place un algorithme qui va redistribuer les données. La méthode de déclenchement (*triggering mechanism*) de l'algorithme de rééquilibrage des données est importante. Elle peut rendre la politique d'équilibrage dynamique inéfficace si la fréquence n'est pas adaptée à l'application. Dans la version SIMD de la machine dictionnaire [Rou93, DFG90], les instructions (insertion, destruction, recherche d'une clef) sont regroupées par paquet pour être diffusées sur l'ensemble des PEs, puis sont exécutées par les PEs concernés. Une fois le traitement terminé les résultats sont renvoyés. Entre deux diffusions de paquet d'instructions une phase de rééquilibrage des entrées du dictionnaire est déclenchée. Le nombre d'entrées contenues sur chaque PE est comparée à une valeur idéale et les entrées en trop sont répartis de proche en proche sur les PEs déficitaires. Dans cet exemple la fréquence de déclenchement de l'algorithme de rééquilibrage dépend du nombre d'instructions dans un paquet.

les machines MIMD

Dans les machines MIMD, on distingue trois classes de machines les multiprocesseurs, les multicomputers, et les systèmes distribués faiblement couplés. Chacune de ces trois classes influence la façon d'aborder le problème du placement. Les entités à placer auxquelles on s'intéresse le plus souvent sont les tâches ou les processus. Dans le cas des systèmes distribués faiblement couplés, c'est le placement de fichiers qui est abordé comme par exemple pour un système de bases de données réparties.

- Les **multiprocesseurs** ont comme particularité d'avoir une mémoire globale et un réseau d'interconnexion pour y accéder. L'algorithme de placement devra s'attacher à limiter les conflits d'accès à la mémoire qui provoque un ralentissement des traitements par attente de disponibilité d'un opérande. La mémoire globale permet de faciliter la gestion de l'état du système, ce qui facilite la mise en place de toutes les stratégies de placement dynamique (voir 1.6.2)

- Pour les **multicomputers**, l'algorithme de placement doit rechercher la solution qui réduise le temps ou le coût d'exécution en favorisant une répartition ou un équilibrage de la charge sur l'ensemble des noeuds. Cette distribution de la charge peut se faire en déplacant ou en dupliquant les données ou encore en faisant migrer le traitement vers les données tout en limitant les sur-coûts de communication dûs aux envois de messages interprocessus sur des noeuds distants.

- Les **réseaux locaux** de stations de travail posent les mêmes problèmes aux algorithmes de placement que les multicomputers, avec toutefois une différence dans le paramétrage des coûts (des temps) pour la communication et la charge des processeurs.

Le problème du placement de fichiers est généralement posé pour les réseaux locaux ou nationaux d'ordinateurs, dans ce cas c'est le coût de communication des requêtes sur les fichiers qu'il convient de limiter, tout en assurant une certaine tolérance aux pannes.

1.3.3 Conclusion sur les architectures

Les machines SIMD qui permettent un parallélisme de données traitent les applications qui nécessitent un traitement lourd en terme de calcul, mais systématique, sur des données qui constituent des structures régulières. Le placement sur cette architecture se limite donc à un placement optimal des données, limitant

les temps d'accès concurrents à la mémoire pour les machines vectorielles pipelines et le nombre d'instructions de communication pour les machines tableaux.

Les machines MIMD qui proposent un parallélisme de traitement, permettent de traiter des problèmes comme la simulation de systèmes complexes, les algorithmes développés dans le domaine de l'intelligence artificielle, et l'implémentation de réseaux neuronaux, qui manipulent des données d'un volume qui peut être important et structurées de façon irrégulière. Le placement doit répondre à plusieurs demandes : une répartition ou un équilibrage de la charge de travail, une limitation du coût de communication inter-noeud pour les multicomputers et les réseaux de stations de travail, une limitation des accès concurrents de la mémoire par les multicomputers.

Dans le problème qui nous intéresse plus particulièrement, les architectures SIMD ne peuvent pas répondre aux types d'applications qui sont développés dans les langages parallèles à objets actifs, car elles développent un parallélisme de traitement qui engendre une évolution dynamique importante qui s'adapte mal à l'architecture SIMD. Dans la suite les architectures que nous utiliserons seront MIMD pour répondre au caractère irrégulier à la fois des structures de données utilisées et des processus développés pour les traiter.

1.4 L'objectif d'une stratégie de placement

L'objectif à atteindre par une stratégie de placement d'une application chargée de mettre en place une chaine de contrôle de fabrication, n'est pas le même que celui qui consiste à limiter le temps de réponse pour une réservation d'un billet d'avion, pour obtenir l'état d'un compte en banque, ou encore le résultat de la détermination du meilleur déplacement d'un cavalier sur un échiquier.

Chacun de ces types d'applications demande une stratégie de placement spécifique en fonction de critères et de contraintes plus ou moins importantes suivant le cas. Dans ce paragraphe nous allons présenter les différents objectifs qui motivent un placement, et pour chacun d'eux, détailler les critères utilisés pour l'atteindre.

1.4.1 Obtenir un coût d'exécution minimal

Obtenir un coût d'exécution minimal est sans aucun doute l'objectif qui a été le plus largement étudié [AP88, MT91, LA87, ST85, MLT82]. La recherche d'un coût d'exécution minimal traduit le but à atteindre de l'une des modélisations permettant de spécifier un ensemble de critères qui vont diriger l'algorithme de placement dans ses choix de distribution de la charge. Ces critères sont intégrés

dans une fonction de coût, dont une formule générique se compose de deux parties :

$$f = \sum_{i=0}^{N-1} \sum_{k=0}^{M-1} g_{ik} x_{ik} + w \sum_{\substack{i,j=0 \\ i \neq j}}^{N-1} \sum_{k,l=0}^{M-1} h_{ijkl} x_{ik} x_{jl}$$

f est en général accompagnée de contraintes permettant d'exprimer la limite en taille mémoire, la limite de charge de calcul, ou des contraintes de redondance (voir 1.4.4) ou des contraintes de terminaison (voir 1.4.3). f n'a un sens que si toutes les tâches décrivent le même cycle de séquences :

- Attente des données;
- Traitement des données;
- Envoi des résultats.

g_{ik} est égal au coût de calcul d'une tâche t_i, d'un processus p_i, ou encore du coût de placement d'une copie d'un fichier c_i sur un noeud n_k;

h_{ijkl} est égal au coût de communication entre une tâche t_i (ou processus p_i) sur un noeud n_k, et une tâche t_j (ou processus p_j) sur un noeud n_l, ou le coût d'une requête d'une tâche t_i (ou processus p_i) sur un noeud n_k, vers une copie de fichier c_j sur un noeud n_l.

N est le nombre de tâches (processus ou fichiers) à placer, M est le nombre de noeuds de la machine parallèle.

$x_{ik} = 1$ si t_i (p_i ou c_i) est sur le noeud n_k , 0 sinon

w permet d'additionner g_{ij} et h_{ijkl} qui ne sont pas obligatoirement exprimés dans la même unité [KM87].

g_{ij}, et h_{ijkl} peuvent être complexes, et permettent d'exprimer les critères que l'on veut satisfaire, ou permettent de s'adapter à l'architecture de la machine parallèle. Voici quelques exemples tirés de [AP88, MT91].

- Pour une architecture homogène avec un bus comme réseau de communication

$$f_0 = \sum_{\substack{i,j \\ i<j}} C_{ij} x_{ij}$$

C_{ij} : coût de communication entre les tâches (processus, fichiers) i et j

$x_{ij} = 1$ si les tâches (processus, fichiers) i et j sont sur des noeuds différents.

Dans cette fonction de coût on cherche à réduire les coûts de communication, le coût d'exécution g_{ik} disparaît car on considère que $\forall k \in P$, P ensemble des noeuds, g_{ik} a toujours la même valeur.

- Pour une architecture homogène avec un réseau de communication ayant une distance variable (voir 1.3.1.2) comme pour la Multicluster II de chez Parsytec

$$f_1 = \sum_{\substack{l,q \\ l \neq q}} \sum_{\substack{i,j \\ i \neq j}} C_{ij} D_{lq} x_{il} y_{jq}$$

D_{lq} : coût de communication unitaire sur le lien entre les noeuds l et q

C_{ij} : quantité de communication entre les tâches i et j

- Pour une architecture hétérogène comme un réseau local il est intéressant de tenir compte de coût d'exécution d'une tâche (processus) en fonction du noeud où il s'exécute [MLT82]

$$f_2 = f_1 + \sum_l \sum_i e_i k_l x_{il}$$

e_i : coût d'exécution de la tâche i

k_l : coût d'utilisation du processeur l

- Même chose que précédemment en ajoutant un coût pour la perte de parallélisme quand on met deux tâches sur le même processeur

$$f_3 = f_2 + \sum_l \sum_{\substack{i,j \\ i \neq j}} e_i k_l x_{il} x_{jl} B_l$$

B_l : coût de la perte de parallélisme pour le noeud l

Les précédentes fonctions de coût considèrent que toutes les tâches peuvent s'exécuter en même temps et dès le lancement de l'application. Cette façon de modéliser le problème du placement permet d'utiliser au mieux les ressources particulières de l'architecture afin d'améliorer le débit du système et cela pour diminuer les temps de réponse moyen (Interrogation , mise à jour d'une base de données), mais ne permet pas d'obtenir le placement donnant un temps de réponse minimal.

Dans [LA87] l'auteur indique une fonction de coût qui permet d'exprimer le fait que les coûts de communication entre deux tâches sont variables dans le temps suivant que le message est attendu ou non par la tâche destination, ou que le message est envoyé en même temps qu'un autre sur le même canal physique. Cependant l'étude se situe dans le contexte particulier d'une architecture homogène, et considère l'ensemble des tâches comme ayant un temps de calcul égal et une exécution synchrone.

1.4.2 Obtenir un temps d'exécution minimal

Pour obtenir le placement qui donne un temps d'exécution minimal, il faut ajouter une nouvelle information par rapport au précédent objectif, il s'agit du graphe de dépendance des tâches, ce graphe permet de mettre en évidence les relations entre les tâches (voir Fig. 1.7).

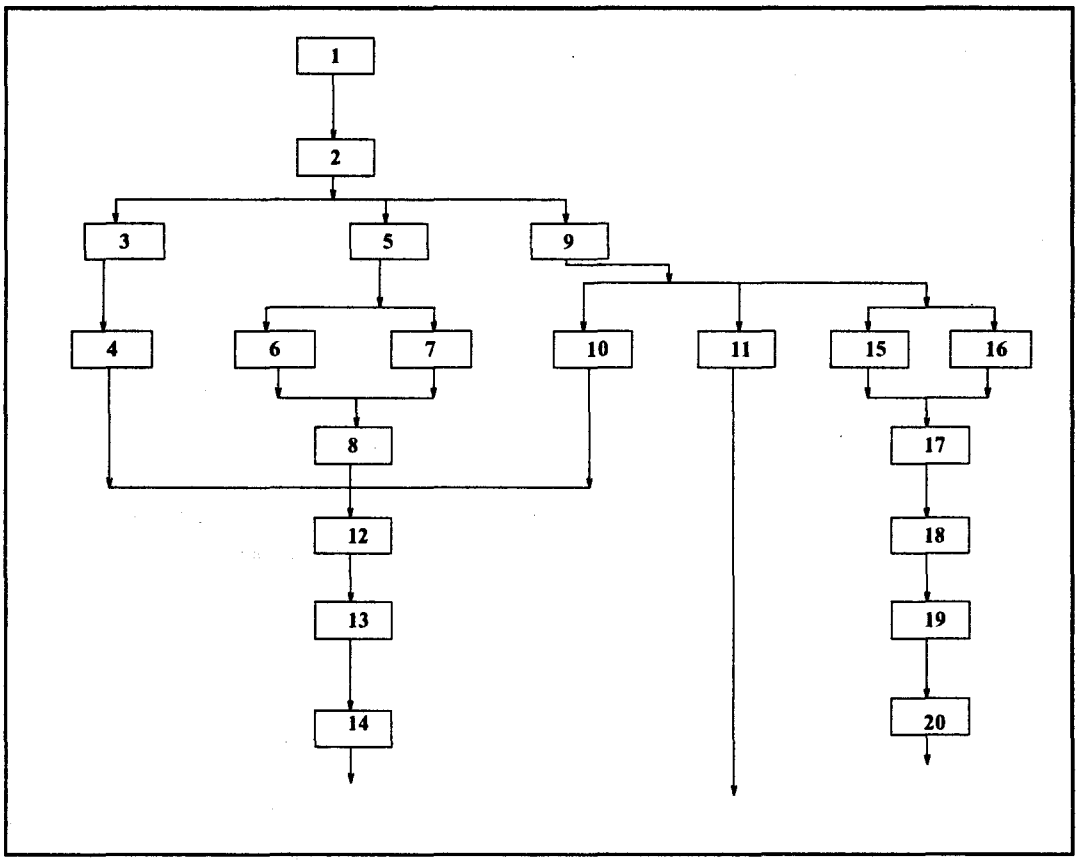


FIG. 1.7 - *Exemple de graphe de dépendances*

Dans [Col91] un programme est un ensemble de tâches indivisibles (attente des données, traitement, envoi des résultats). Une tâche ne peut pas s'exécuter avant d'avoir reçu les informations des tâches qui la précédent. Chaque échange de message entre deux tâches prend un certain temps non négligeable, qui dépend de la taille du message et de la distance entre les noeuds qui exécutent les tâches, si les tâches sont sur le même noeud le temps de communication est considéré comme négligeable. Les critères qui permettent de déterminer le placement ayant un temps d'exécution minimal est :

- **La durée d'exécution des tâches** dans une architecture homogène, p_r est la durée de la tâche r

- **Les temps de communications** $v_{r,s}$ sont positif si $n_r \neq n_s$, nuls sinon
- **Les contraintes de précédence** G est un graphe, un arc (r, s) de G correspond à une dépendance de la tâche r vers la tâche s. L'arc est valué par $v_{r,s}$

Dans ce cas la recherche du placement revient à trouver un ordonnancement de tâches de durée globale minimale ce qui revient à obtenir un chemin critique dans le graphe et chercher la valeur minimale de son temps d'exécution [CC88].

1.4.3 Prise en charge des applications temps réel

Dans les applications temps réel comme le contrôle de processus, certaines tâches doivent être terminées avant une certaine date. Dans [Ma84] les dates limites de terminaison d'un ensemble de tâches sont données comme des contraintes, que l'algorithme de placement doit obligatoirement satisfaire.

Dans [CHLE80] la contrainte est donné par

$$\sum_i u_i x_{i,k} \leq T_k, \ k = 1, \ldots, n$$

avec u_i qui représente le temps d'exécution du module M_i, et T_k la contrainte de temps limite pour la terminaison de l'exécution de l'ensemble des modules sur le noeud P_k.

1.4.4 La tolérance aux pannes

La tolérance aux pannes est introduite dans les algorithmes de placement par l'ajout d'une contrainte encore appelée technique de redondance des processus [BT83].

$$\forall i \in P : \sum_l x_{il} = r_i$$

Cette contrainte indique qu'il faut r réplications de la tâche i, chaque copie de la tâche est placée sur un noeud différent.

Le projet Paralex [BAAD91], dont le but est de fournir à l'utilisateur un environnement de programmation pour l'utilisation d'un réseau de stations de travail comme machine parallèle, permet à l'utilisateur de définir un niveau de tolérance aux pannes. Le compilateur crée autant de copie du code que nécessaire pour garantir le niveau de tolérance aux pannes, à la suite de quoi, un des buts de l'algorithme de placement est de placer chaque copie sur un noeud différent.

1.4.5 Conclusion sur l'objectif d'une stratégie de placement

Dans ce paragraphe nous avons présenté les différents objectifs d'un placement.

La minimalisation d'une fonction de coût est généralement utilisée, il y a en cela plusieurs raisons. La première est qu'elle permet dans les cas simples d'obtenir un système d'équations linéaires (en transformant les termes non linéaires) dont on connaît des algorithmes permettant de trouver les solutions (algorithme du simplex par exemple). D'autre part cette modélisation permet aussi d'intégrer facilement des contraintes définies par l'utilisateur, mais aussi des contraintes spécifiques à une architecture cible, ce qui est moins facile dans d'autres modélisations (voir 1.6).

La fonction de coût ne permet pas d'obtenir, dans le cas général, un placement assurant un temps minimal d'exécution, il faut pour cela introduire une nouvelle donnée, le graphe de dépendances entre les tâches. Ce qui aboutit à la résolution d'un problème d'ordonnancement de tâches d'une durée d'exécution globale minimale.

Les objectifs spécifiques à un type d'environnement comme les applications temps réel ou la définition d'un niveau de tolérance aux pannes sont modélisés par l'ajout de contraintes dans les précédentes formulations.

L'utilisation de ces fonctions de coût peut se faire à des niveaux différents du développement de l'application (voir paragraphe suivant). Certaines très complexes à calculer sont réservées pour la recherche du placement statique de l'application (par exemple pendant l'étape de compilation). D'autres plus élémentaires peuvent être utilisées au chargement de l'application pour s'adapter à l'architecture de la machine cible et aussi pendant l'exécution de l'application.

1.5 Localisation dans le temps des algorithmes de placement

Le placement est qualifié de statique quand il est fait au moment de l'analyse de l'application, et il est qualifié de dynamique quand il est fait au cours de l'exécution de l'application.

Or le placement intervient aussi quand le programmeur donne explicitement dans le source de son application la localisation des structures de données, mais aussi lorsqu'il fait le découpage de son algorithme en tâches autonomes, la détermination du grain des tâches est un problème difficile [KL88a]. Le placement peut être fait aussi lors du chargement des tâches. On peut considérer quatre étapes

(voir Fig. 1.8) importantes pour le placement d'une application :

- Le développement de l'application
- La compilation
- Le chargement
- L'exécution

Pour chacune de ces étapes nous allons présenter les informations et traitements qui peuvent apporter des éléments qui vont permettre de résoudre le problème du placement, en donnant leurs avantages et inconvénients.

1.5.1 Placement à la création de l'application

1.5.1.1 Découpage de l'application

Le découpage de l'application peut avoir une grande importance sur son temps d'exécution. Dans [KL88a], les auteurs déterminent de façon automatique la taille idéale d'une tâche permettant d'obtenir un temps d'exécution optimal. L'application est représentée par un graphe sans cycle orienté $pg(n, e)$, avec n l'ensemble des opérations atomiques, et e l'ensemble des relations de précédence entre les opérations. Le problème du regroupement (*clustering*) est de trouver un graphe de tâches $TG(N, E)$ avec N_i une tâche qui sera exécutée sur un noeud de la machine (N_i peut contenir une opération atomique ou un ensemble de n_i opérations de n), tel que le temps d'exécution de $pg(n, e)$ soit minimal.

La détermination du contenu des tâches N_i est faite en quatre étapes :

- La **construction du graphe des opérations atomiques** qui fournit un temps d'exécution pour chacune d'elle, et un temps de communication entre elles.
- La **détermination d'un ordonnancement des opérations atomiques** qui est faite à l'aide d'un algorithme développé par les auteurs : DSH (*Duplication scheduling heuristic*)
- Le **regroupement des opérations atomiques en tâches** qui réduit les temps de communication, certaines opérations peuvent être dupliquées et se trouver dans des regroupements différents.
- La **génération des modules** est faite à la compilation et repose sur le résultat de l'étape précédente.

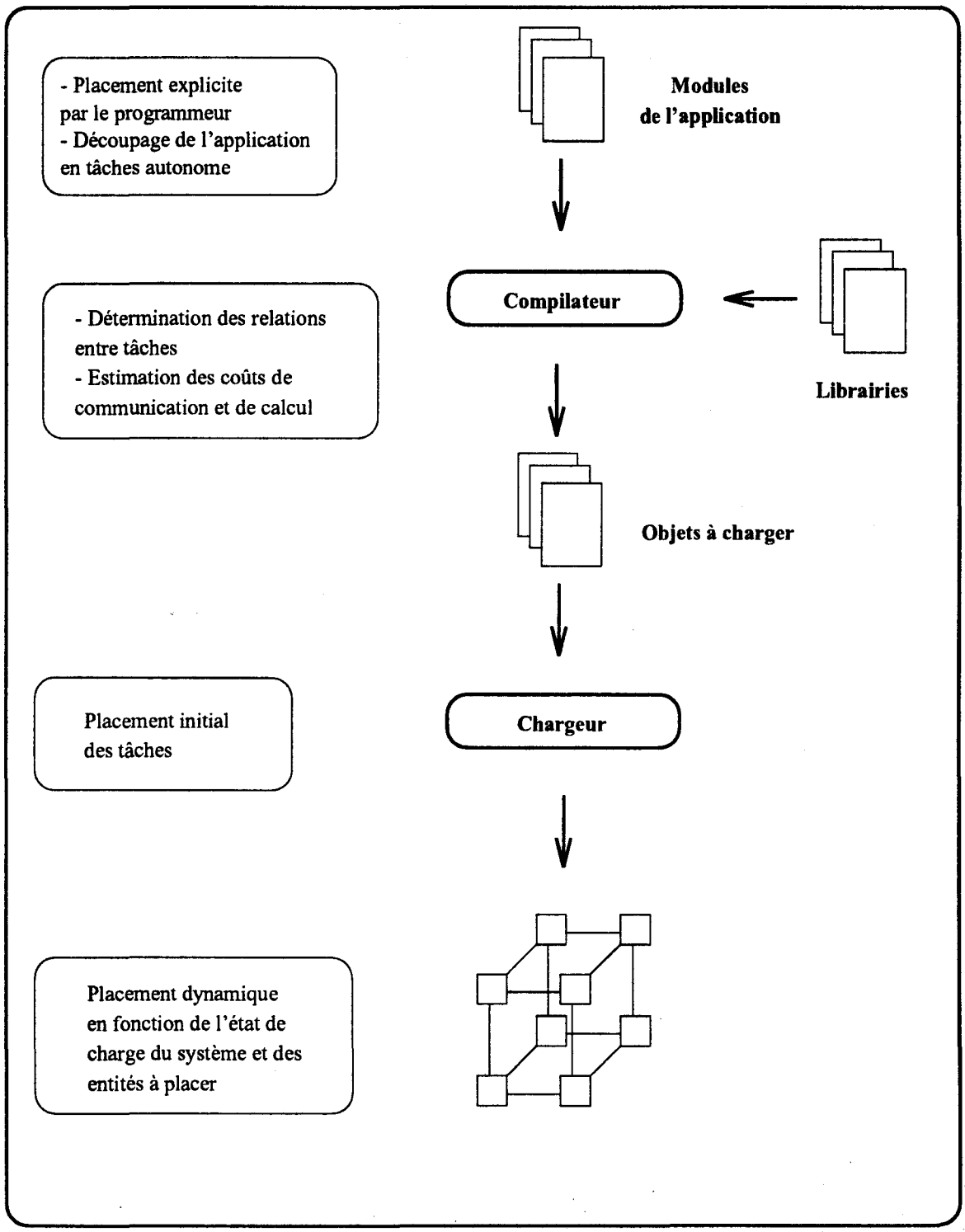


FIG. 1.8 - *Les étapes de placement d'une application*

Le découpage de l'application peut aussi être le résultat des spécifications de contraintes structurelles ou de contraintes de sécurité, par exemple dans le cas d'applications utilisant des ressources particulières disponibles sur un nombre limité de noeuds, ou dans le cas d'utilisation de données sensibles qu'il ne faut pas dupliquer. Ces spécifications entrent généralement en conflit avec l'objectif des algorithmes de placement, elles sont donc à éviter dans les cas où cela est possible.

Lorsque l'entité de base à manipuler peut être à grain fin comme dans les langages à objets du type acteur, il peut y avoir une phase de regroupement du code afin de limiter le coût de communication entre acteurs qui communiquent beaucoup. Cette phase de regroupement est difficile à identifier de façon automatique du fait de l'évolution dynamique des relations entre acteurs. Dans ce cas seul l'utilisateur qui connaît son application peut introduire des directives de regroupement en module.

1.5.1.2 Recommandation de l'utilisateur

Dans certains langages de programmation, il existe des instructions permettant aux programmeurs de faire du placement explicite. Dans POOL2/PTC [AH91] le programmeur influence le placement des objets (qui sont des processus par défaut) à l'aide de *pragmas.* Ces *pragmas* donnent le placement d'un objet sur un noeud physique relativement à un autre objet. Dans le langage *ORCA* [BKT90]ou dans PSather [MFL93], le programmeur peut spécifier un numéro de noeud. Dans les langages type *CSP* [Hoa78] comme OCCAM le programmeur doit explicitement indiquer le numéro physique du noeud sur lequel on va exécuter une procédure.

Les langages explicites à parallélisme de données [Mar92] disposent de nombreux mots clefs permettant aux programmeurs de proposer une distribution des données sur des processeurs virtuels, un processeur virtuel correspondant généralement à un processeur phisique. Le langage HPF qui est conçu comme une extension de Fortran 90 intégre des mots clefs permettant à l'utilisateur de donner des recommandations de distribution de données sous forme de directives (ALIGN, DISTRIBUTE BLOCK, DISTRIBUTE CYCLIC, etc ...).

1.5.2 A l'analyse du source

Le placement à la compilation repose sur les informations obtenues suite à l'analyse du source d'une application par le compilateur.

Le compilateur des langages destinés à exploiter le parallélisme de données traduit les instructions de placement explicites des données pour la machine cible.

Suite à une analyse de la structure des données il parallélise les opérations indépendantes (traitement en parallèle sur l'ensemble des scalaires d'un vecteur par exemple) ce qui n'est pas sans poser des problèmes pour une adaptation à une architecture particulière [Paz93].

Pour les architectures permettant un contrôle MIMD et quand le langage ne permet pas de créer dynamiquement des processus, le compilateur est à même de déterminer les relations de communication et de précédence qui existent entre les tâches d'une application. Par une analyse plus fine il peut quantifier la complexité d'une tâche en terme d'instructions élémentaires ce qui permettra de prévoir la durée de son exécution. Le volume des données échangées est cependant plus délicat à évaluer et dépend le plus souvent des données à traiter. Ces informations peuvent permettre d'effectuer une phase de découpage de l'application en regroupant les entités qui travaillent ensemble. Dans le cas d'une recommandation faite par l'utilisateur, la phase de regroupement peut être confirmée ou infirmée, le traitement de ce genre de problèmatique n'est pas simple à resoudre, elle peut être gérée par l'émission d'un message d'erreur non bloquant indiquant le problème. L'ensemble des algorithmes de placement statique repose sur les informations fournies par le compilateur. Le résultat est basé sur une description de la topologie de la machine cible ou bien aucune topologie n'est imposée *a priori* et il peut aussi fournir la topologie idéale pour l'exécution de l'application (c'est le cas pour les machines avec une architecture reconfigurable dynamiquement).

Dans le cas des langages disposant d'instructions permettant une création dynamique, le compilateur ne peut pas en général déterminer les relations qui relient les tâches. Il peut néanmoins indiquer des tendances sur le graphe de déclenchement direct des tâches. L'analyse de la complexité se fait dans les mêmes conditions que précédemment. Dans ce cas un placement statique ne peut pas garantir une utilisation optimale de la machine cible vis à vis du but que l'on voulait atteindre, le compilateur ne pourra fournir que les informations statiques qui permettront d'effectuer un placement initial du code des tâches sur la machine cible.

1.5.3 Au chargement de l'application

Cette étape peut se réduire à un simple chargement du code des tâches de l'application sur les noeuds de la machine cible, quand l'algorithme de placement statique a utilisé la topologie de la machine cible. Cette étape du placement permet de faire une adéquation entre les résultats du placement statique obtenus grâce au compilateur et l'architecture de la machine cible. Le problème consiste à faire un recouvrement du graphe des tâches de l'application sur le graphe que représente la topologie de la machine. En cas de panne d'un noeud l'algorithme de chargement de l'application devra pouvoir trouver une solution de rechange

qui ne pénalise pas l'exécution de l'application, quand c'est possible. Il s'agit le plus souvent de faire un regroupement des tâches qui sont plus nombreuses que le nombre des noeuds pour réduire les coûts de communication ou de dupliquer les tâches qui représentent une grosse activité sur les noeuds libres quand le nombre de tâches est inférieur au nombre de noeuds.

Les informations fournies par l'utilisateur et le compilateur doivent permettre une adaptation à l'architecture de la machine effectivement utilisée, dans le cas contraire on peut aboutir à des situations où la machine est très mal utilisée voir même interdire l'exécution de l'application pourtant possible.

1.5.4 A l'exécution

Le placement à l'exécution a pour but de faire de l'équilibrage ou de la répartition de charge sur l'ensemble des noeuds, il est basé sur l'état courant du système. La grosse difficulté se situe dans l'implantation d'une stratégie qui ne perturbe pas trop l'exécution de l'application, par l'ajout d'un coût de communication et de calcul engendré par le placement lui-même. Un aperçu des stratégies de placement dynamique sera examiné dans le paragraphe 1.6.2.

Il existe aussi des algorithmes de *rééquilibrage de charge* pour machines SIMD (voir paragraphe 1.3.2). Ces algorithmes se distinguent des algorithmes de placement pour machines MIMD, par le fait que l'opération de répartition ou d'équilibrage de charge doit être fait par l'ensemble des noeuds en même temps. Leur fonctionnement alterne une phase d'exécution de l'application avec une phase de rééquilibrage. Le problème est de déterminer la fréquence de déclenchement des phases de rééquilibrage qui si elles sont trop rapprochées pénalisent l'exécution de l'application sans apporter d'amélioration et inversement si elles sont trop espacées pénalisent l'exécution pour cause de mauvaise répartition. La phase de rééquilibrage est déclenchée [PFK93] lorsqu'un certain seuil (représentant le nombre des noeuds inactifs par exemple) est atteint. La valeur du seuil peut être statique dans certains algorithmes ou dynamique.

Les informations fournies par l'utilisateur ou par le compilateur ont été le plus souvent utilisées dans le passé comme des directives de placement obligatoire par les algorithmes de placement dynamique. On essaye de plus en plus d'intégrer ces informations comme un paramètre dans les stratégies de détermination du noeud lors d'un rééquilibrage [Fol93].

1.5.5 Conclusion

Le problème de placement d'une application sur architecture parallèle peut être abordé à tous les niveaux du cycle de son développement. Le découpage de

l'application en tâches puis en modules aura des conséquences sur son exécution. Dans la phase de compilation il est important que soit automatisée, quand cela est possible, la recherche des relations de dépendance et de communication entre les tâches, de même qu'une évaluation de la complexité de chaque tâche, pour pouvoir faire un placement statique ou aider l'algorithme de chargement de l'application sur la machine parallèle. Le placement dynamique est envisagé lorsque l'application a un comportement qui évolue au cours de son exécution, ou lorsque l'état de la machine (utilisation en multi-utilisateurs) varie au cours de l'exécution.

L'intégration des informations obtenues à chaque niveau du développement d'une application dans le niveau suivant n'est pas sans poser quelques problèmes. En effet les recommandations fournies par le programmeur peuvent être mise en défaut par le compilateur lors de l'analyse du source. Le placement initial statique fait par l'analyseur peut être remis en cause par l'étape de chargement qui doit s'adapter aux pannes éventuelles. Enfin le placement dynamique peut recevoir des recommandations multi-niveaux qui entrent en conflit avec l'état courant du système. Une solution pourrait être d'intégrer un coefficient de vraisemblance à chaque information, ou encore le traitement de l'ensemble des données par un système expert qui ferait un choix en fonction de la base de faits et de régles génériques.

1.6 Les différents modèles d'algorithmes de placement

1.6.1 Placement statique

Les algorithmes de placement statique se situent entre l'étape de compilation et l'étape de chargement de l'application sur la machine cible. Les différents modèles nécessitent de connaître l'ensemble des tâches qui sont à placer et les relations qui les lient. Pour chacune des tâches on dispose d'une donnée permettant de les comparer en terme de complexité. Cette donnée représente un coût d'exécution. Chaque relation entre les tâches est valuée par une donnée qui représente le volume d'échange entre les tâches. Ces données sont statiques et connues avant le lancement de la résolution du placement.

Les méthodes d'obtention des données sont variables, elles peuvent être le résultat d'une :

- analyse faite dans l'étape de compilation;
- modélisation analytique de l'application comme les queues d'attentes, un

réseau de Petri stochastique: les informations sont obtenues rapidement, mais ne sont pas très précises;

- simulation qui évite de disposer d'une machine parallèle dans la phase de développement d'une application;
- évaluation des exécutions précédentes.

La qualité des informations obtenues varie suivant la méthode utilisée, elle varie généralement avec le temps d'exécution des algorithmes utilisés pour les obtenir. Le projet ALPES [KP92] qui a pour but de fournir un environnement d'évaluation des performances d'une application parallèle, modélise une application par un graphe de tâches valué obtenu à partir d'un programme parallèle synthétique dont on analyse le comportement.

L'environnement PSEE [LSS92] permet la simulation et l'évaluation des systèmes parallèles.

Dans le projet Visage [RZZ93] les applications sont modélisées par des graphes.

Le choix d'un algorithme de placement statique peut se faire en fonction de la qualité du placement que l'on veut obtenir. Une application qui a une fréquence d'utilisation faible pourra se contenter d'un placement approximatif, une application très souvent utilisée devra bénéficier d'une étude de son placement plus approfondie. Cependant le choix d'un placement optimal est critiquable dans le sens où sa détermination repose sur des constantes alors qu'en général (sauf pour les applications temps-réel) , elles sont fonction des paramètres d'entrées.

Les algorithmes de placement statique peuvent être classifiés dans l'une des trois approches suivantes (voir [MT91, CT93, Rou93] pour une synthèse compléte) selon les outils mathématiques qu'ils utilisent :

- la théorie des graphes;
- la programmation mathématique;
- l'utilisation d'heuristiques.

1.6.1.1 La théorie des graphes

Pour modéliser l'architecture de la machine cible on utilise un graphe non orienté $G_m = (V_m, E_m)$. V_m l'ensemble des sommets du graphe, représente les noeuds de la machine cible, et E_m l'ensemble des arcs du graphe, indique les liaisons entre les noeuds. Le graphe n'est pas orienté car on considère que les liaisons sont bidirectionnelles, il peut être valué dans le cas d'une architecture

hétérogène, dans ce cas V_{m_i} est valué par son coût d'utilisation, $E_{m_{ij}}$ est valué par le coût d'utilisation de la communication entre V_{m_i} et V_{m_j}.

De la même façon l'application est modélisée par un graphe non orienté $G_t = (V_t, E_t)$, avec V_t l'ensemble des tâches de l'application, et E_t l'ensemble des relations de communications entre les tâches. Les arcs et les sommets sont valués respectivement par le volume de communications entre les tâches et par le coût d'exécution (qui peut être un temps ou introduire un facteur économique).

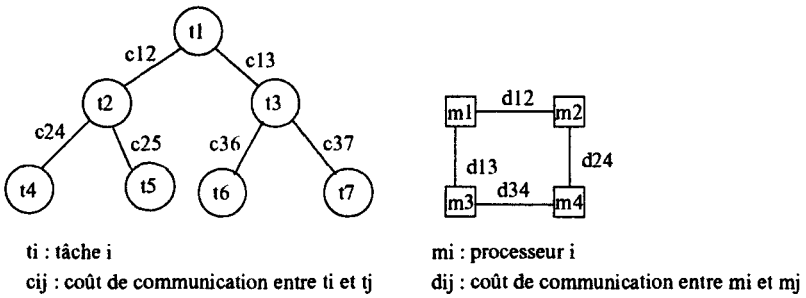


FIG. 1.9 - *Modélisation d'une application et d'une architecture*

Le problème du placement revient à trouver une relation $r : V_m \rightarrow V_t$ qui alloue un processeur l de V_m à chaque processus i de V_t : $l = r(i)$. Cette relation n'existe que dans le cas où il existe une projection du graphe G_m vers le graphe G_t qui garantit que deux processus voisins sont alloués sur deux processeurs voisins. Dans [Bok81a], l'auteur se place dans le cas où le nombre de tâches est inférieur ou égal au nombre de noeud, le meilleur placement est celui qui minimise le nombre de chemins indirects. Pour cela l'auteur définit la *cardinalité* qui représente le nombre d'arcs du graphe des tâches qui coïncident avec les arcs du graphe des noeuds. L'algorithme de recherche du meilleur placement est basé sur une heuristique décrite dans le paragraphe 1.6.1.3.

Une autre modélisation est celle donnée dans [Sto77, Lo84], les sommets du graphe représentent à la fois les tâches et les noeuds. Un arc reliant deux tâches i et j est valué par un coût de communication c_{ij}. Un arc reliant une tâche i à un noeud q est valué par w_{iq}

$$w_{iq} = \frac{1}{n-1} \sum_{r \neq q} e_{ir} - \frac{n-2}{n-1} e_{iq}$$

avec e_{iq} représentant le coût d'exécution de la tâche i sur le noeud q.

Le problème du placement revient à trouver la partition de coupe minimale du graphe, chaque partition contenant un noeud et un ensemble de tâches qui sont placées sur celui-ci (voir Fig.1.10).

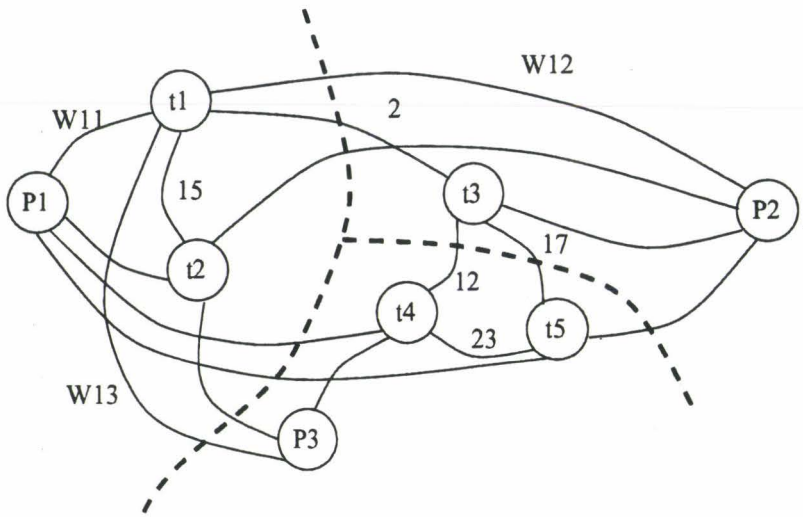


FIG. 1.10 - *Exemple de placement par partitionnement*

Coût d'exécution			
	p_1	p_2	p_3
t_1	3	11	23
t_2	1	∞	9
t_3	14	10	21
t_4	15	6	8
t_5	∞	2	7

Coût de communication					
	t_1	t_2	t_3	t_4	t_5
t_1	0	15	2	0	0
t_2	15	0	0	0	0
t_3	2	0	0	12	17
t_4	0	0	12	0	23
t_5	15	0	17	23	0

Ce modèle ne peut être étendu à un nombre quelconque de noeuds n, pour $n \geq 3$, le problème devient NP-complet, les temps de calcul explosent de manière exponentielle. De plus il considère que les coûts de communication entre les noeuds sont identiques.

La modélisation par la théorie des graphes permet difficilement d'intégrer des contraintes spécifiques à une architecture ou à l'application, et rend le problème NP-complet, c'est pourquoi on lui préfère la modélisation par programmation mathématique.

1.6.1.2 La programmation mathématique

La méthode modélise le problème du placement comme un problème d'optimisation combinatoire. Le problème se présente sous la forme d'une fonction de coût à minimiser (1.1), en tenant compte de contraintes spécifiques (1.2).

$$S = \min f(X_1, X_2, \ldots, X_k) \tag{1.1}$$

$$g_i(X_1, X_2, \ldots, X_k) \leq B_i \tag{1.2}$$

avec X_i qui sont des variables bivalentes, f et G_i sont des fonctions polynomiales en X_i à coefficients constants.

Les B_i sont des constantes et S est la solution du problème.

De nombreuses fonctions de coûts ont été utilisées, nous en avons présenté quelques unes dans le paragraphe 1.4.1.

Le parcours de l'espace des solutions utilise différentes méthodes comme la séparation et l'évaluation successives [MLT82], l'algorithme du A^* [ST85], la programmation dynamique [Bok81b], mais aussi par des heuristiques (voir paragraphe 1.6.1.3).

1.6.1.3 Les heuristiques

Comme nous l'avons vu précédement la recherche d'une solution optimale au problème du placement est généralement un problème NP-complet quand la taille du problème devient trop grand. De plus les données sur lesquelles se base l'algorithme sont généralement sujettes à caution. La solution qui consiste à déterminer une *bonne* solution au problème placement est très souvent retenue car elle est moins coûteuse en temps, en espace, et répond en général assez bien au problème posé. Les heuristiques présentées sont des fonctions polynomiales qui n'examinent pas la totalité de l'espace de solution. Le principe est donc d'examiner un échantillon représentatif de cet espace, et de retenir la meilleure solution.

Il existe deux types d'heuristiques :

- **les algorithmes gloutons** qui à partir d'une solution de départ partielle, cherchent à l'étendre à chaque pas de l'algorithme, en excluant la possibilité de remettre en cause les allocations antérieures.

- **les algorithmes itératifs** qui sont initialisés avec une solution complète qu'ils essayent d'améliorer à chaque itération. La solution de départ est en général une solution aléatoire, et à chaque itération on effectue un échange de placement de processus, cet échange est validé si il diminue la fonction de coût f utilisée.

  Dans [Bok81a] (voir paragraphe 1.6.1.1) l'heuristique mise en place pour examiner l'ensemble des solutions est un algorithme d'échange de placement entre deux processus, si l'échange augmente la *cardinalité* alors le placement est conservé. L'algorithme s'arrête quand il n'y a plus d'amélioration.

Les algorithmes de type gloutons et itératifs donnent des résultats bons en moyenne, mais peuvent être très mauvais dans des cas particuliers où ils restent dans un domaine de solutions sub-optimales. Les **algorithmes du recuit simulé** [BM88] ou les **algorithmes génétiques** [WSB90] et plus récemment la recherche

tabou [Len93] permettent de sortir de ces solutions sub-optimales et convergent vers une meilleure solution, cela au prix d'un temps de calcul plus important.

1.6.1.4 Conclusion sur le placement statique

La modélisation du problème de placement statique aboutit à des algorithmes d'une complexité qui est équivalente à celle de la classe des problèmes NP-Complet quand le nombre de noeuds et le nombre de tâches considérés devient grand ($n \geq 3$). La solution retenue consiste à utiliser des heuristiques qui parcourent l'espace de solutions en retenant la meilleur solution de l'échantillon examiné. La qualité de la solution obtenue par rapport à l'optimal est liée au coût d'exécution. Cependant il ne faut pas oublier que la résolution du problème repose sur des données constantes alors que dans la réalité ces données sont fonction des paramètres d'exécution de l'application. L'obtention d'une *bonne* solution dans un temps raisonnable est donc préférable à une solution optimale.

1.6.2 placement dynamique

Les algorithmes de placement dynamique sont utilisés au cours de l'exécution de l'application que l'on veut placer. Leur mise en oeuvre représente un coût qui vient s'ajouter au coût d'exécution de la ou les applications. Les gains apportés en terme d'utilisation des ressources sont généralement intéressants, par contre les gains en terme de temps de réponse peuvent être décevants compte tenu du sur-coût engendré par la recherche d'une bonne répartition. C'est pourquoi les algorithmes de placement dynamique sont utilisés lorsque les algorithmes de placement statique sont incapables de fournir une bonne solution, c'est à dire quand le nombre de tâches n'est pas connu *a priori*, que les relations de communication évoluent dynamiquement, et que le coût d'exécution d'une tâche est imprévisible.

Les algorithmes, hérités de l'intelligence artificielle, qui parcourent un espace de solutions dont la forme n'est pas connue avant l'exécution, A* font partie des applications où le placement statique n'est pas capable d'intervenir. Les applications basées sur une programmation objets ou acteurs pour leurs taux élevés de création dynamique, et un graphe de communication qui évolue dynamiquement, ne peuvent pas être réparties statiquement.

Sur les architectures qui ne sont pas dédiées à une application particulière, mais partagent un ensemble de ressources au travers d'un système distribué faiblement couplé, le placement d'une application doit tenir compte de l'état courant de l'ensemble des noeuds au moment de l'exécution.

Enfin les algorithmes de placement dynamique permettent une adaptation en cours d'exécution suite à une panne sur l'un des noeuds ou suite à une reconfigu-

ration des liaisons de communication entre noeuds.

Il existe de nombreux algorithmes de placement dynamique, chacun d'entre eux utilise le même mécanisme de fonctionnement (voir Fig. 1.11).

- Evaluation de la charge locale d'un site;

  Permet d'établir un point de comparaison entre les différents sites.

- Echange de la charge entre les sites;

  Permet d'établir un état global de la machine par un ensemble d'informations sur la charge locale des sites.

- Fréquence d'utilisation de l'algorithme de placement dynamique et choix du ou des processus à transférer;

  Détermine à quel moment est déclenché l'algorithme de placement dynamique, et à quelles tâches il va s'intéresser.

- Choix du site qui va accueillir le processus à transférer.

  Permet d'établir des critères de choix du site qui va accueillir le processus à transférer en fonction de la charge, de la distance, des particularités d'un site.

Nous allons maintenant proposer une classification des algorithmes de placement dynamique. Pour cela nous allons examiner l'éventail des possibilités pour chacune des composantes proposées dans la figure 1.11 de manière indépendante vis à vis des autres. Un algorithme de placement dynamique donné sera alors le résultat d'une sélection pour chacune des composantes.

1.6.2.1 Calcul de la charge par noeud

Les algorithmes de placement dynamique sont basés sur une évaluation de la charge sur un noeud. Le calcul peut privilégier différents aspects de la charge [FZ86], avoir une complexité et une fréquence de modification variable.

Propriétés d'un indicateur de charge

Dans [FZ86, FZ87] les auteurs indiquent les propriétés importantes que doit avoir un *indicateur de charge*, et qui sont reprises dans [BSS91] :

1. Estimer la charge courante sur le noeud
2. Estimer la charge sur le noeud dans un futur proche

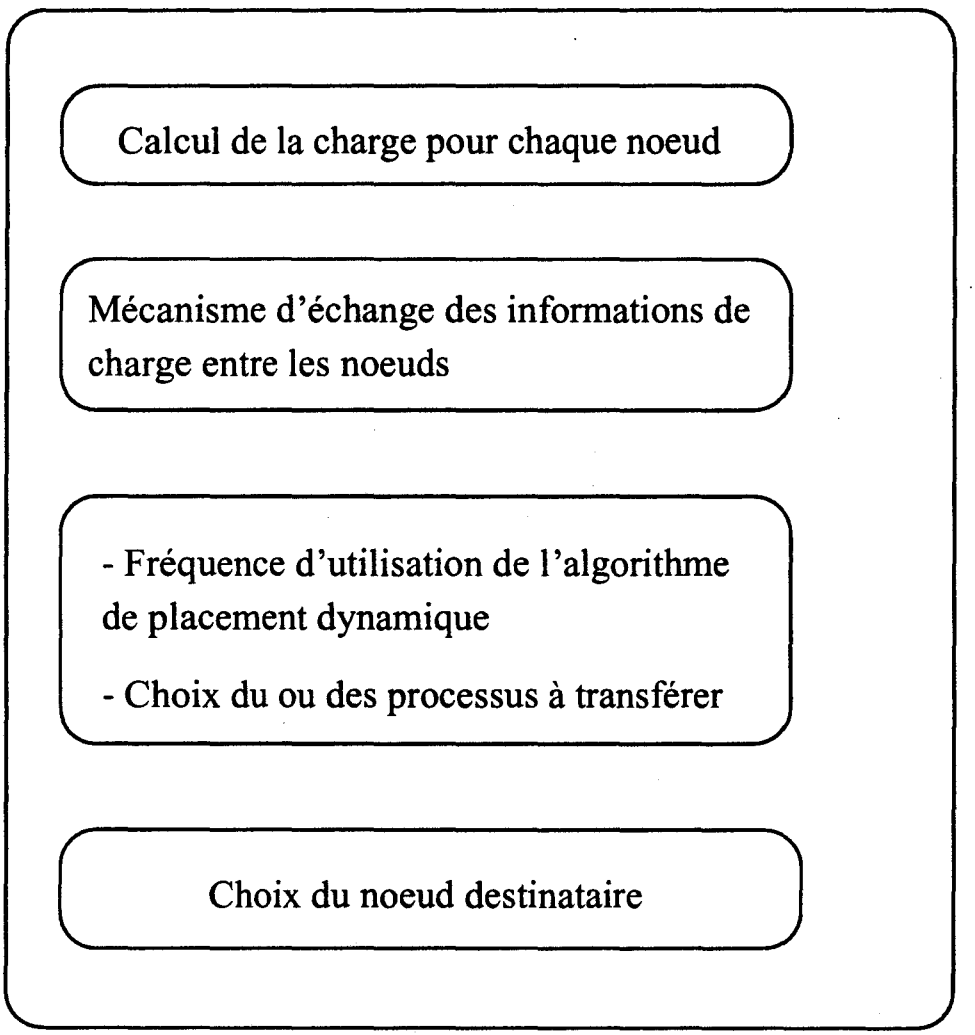


FIG. 1.11 - *Structure d'un algorithme de placement dynamique*

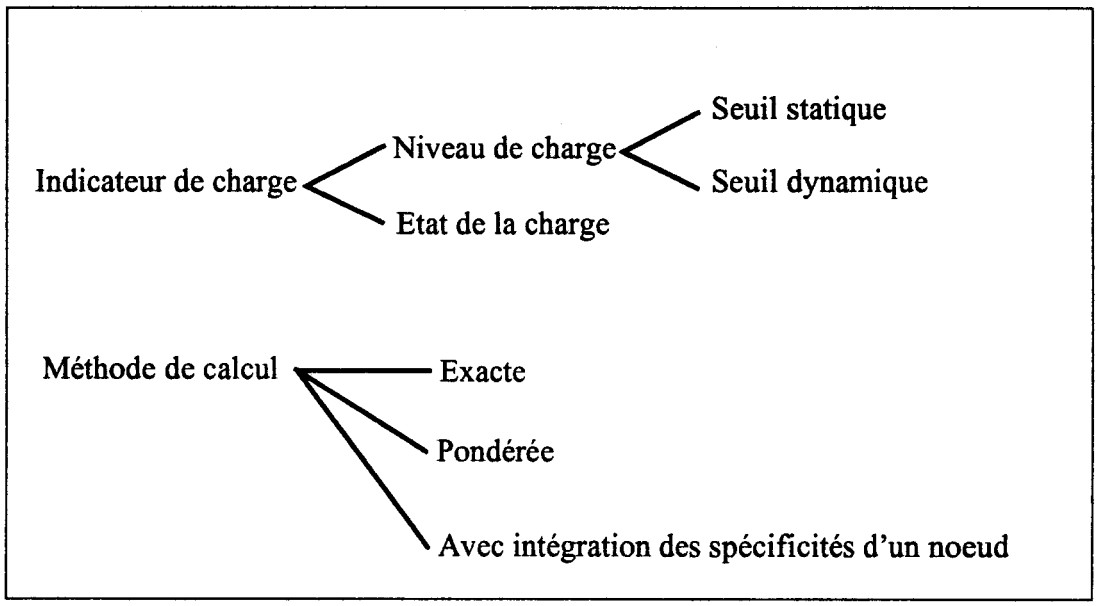


FIG. 1.12 - *Présentation des indicateurs de charge*

3. Gommer les variations de courte durée

4. Prendre en compte les spécificités d'un noeud (utilisation d'une ressource particulière)

Informations pertinentes d'un indicateur de charge

Pour faire une estimation de la charge sur un noeud, il faut déterminer les éléments que l'on considère comme caractéristiques pour son calcul. On peut citer les informations suivantes :

- le nombre de processus présents; c'est-à-dire les processus en attente de la CPU, ou bloqués en attente d'un événement. En effet, il y a une forte corrélation entre la charge future que va avoir à gérer le site et le nombre de processus présents.

- le nombre de processus prêts. Cette fois on considère que les processus en attente n'interviennent pas dans la charge future d'un site.

- le nombre de messages en attente de traitement. Dans les applications basées sur la programmation objet où les processus communiquent par messages, l'utilisation de la taille de la liste des messages à traiter est en forte corrélation avec la charge future qui va être introduite pour traiter les messages.

Dans les réseaux de stations de travail on exprime la charge en terme de taux.

- le taux d'utilisation de la CPU;
- le taux d'occupation de la mémoire;
- le taux d'accès aux entrées / sorties.

qui correspondent à l'information obtenue par le système d'exploitation de la station.

La détermination d'un indicateur de charge n'est pas aussi simple qu'on pourrait le penser. Une application composée de processus qui occupent beaucoup d'espace mémoire et qui s'exécute sur une architecture de machine avec un espace mémoire réduit sera bien placée par un indicateur de charge qui tiendra compte du taux d'occupation de l'espace mémoire. Une application qui consomme beaucoup de temps CPU sera bien répartie par un indicateur de charge qui prend en compte le taux d'occupation de la CPU. Si les deux applications doivent s'exécuter en même temps le choix d'un indicateur de charge n'est pas simple.

La détermination d'une grandeur étalon "charge élémentaire" permettant de comparer les charges sur une machine composée de sites hétérogènes est aussi un problème non trivial. Certains auteurs [Fol93, TL89, Zho88] expriment la charge locale par un ensemble d'états de charge que peut avoir un site. Dans [Zho88] l'auteur utilise un mécanisme à double seuil permettant d'exprimer la charge d'un site par trois états *Libre, Actif, Chargé.* L'indicateur passe d'un niveau à un autre en fonction de deux seuils : *Seuil_Bas, Seuil_Haut.*

debut
  si $0 \leq Charge \wedge Charge < Seuil\_Bas$ **alors**
    $IC \leftarrow Libre$
  sinon
    si $Charge > Seuil\_Bas \wedge Charge < Seuil\_Haut$ **alors**
      $IC \leftarrow Actif$
    sinon
      $IC \leftarrow Chargé$
    fsi
  fsi
fin

Evolution dynamique de la valeur des seuils

Un inconvénient de la méthode à double seuil est que si les valeurs de *Seuil_Bas* et *Seuil_Haut* n'évoluent pas en fonction de la charge globale du système on

peut aboutir à des situations où un site va refuser du travail en provenance de l'extérieur sous prétexte qu'il est dans un état de charge *Chargé* alors que les autres sites ont une charge locale bien supérieure. Une modification dynamique de ces valeurs au cours de l'exécution est souhaitable pour prendre en compte l'évolution de la charge globale sur la machine, mais pose des problèmes dans un environnement distribué et doit répondre aux questions classiques suivantes:

- Qui décide de changer la valeur des seuils?

  Par exemple:

  - Un site Chargé qui ne trouve pas de site pour accueillir les nouveaux processus.
  - Un site Libre qui n'a pas reçu de demandes de prise en charge d'un nouveau processus depuis un certain temps.

- Quand décide-t-on de faire changer la valeur des seuils?

  Pour limiter les modifications intempestives et trop fréquentes, le site ne devra pas avoir reçu une demande récente de modification de la valeur des seuils.

- Comment prévenir l'ensemble des sites?

  Par diffusion de la nouvelle valeur des seuils vers les autres sites

Obtention des informations pour le calcul de l'indicateur de charge

Une fois déterminée l'information pertinente, et une manière de l'exprimer pour qu'elle soit comprise par l'ensemble des sites, il faut mettre en place un mécanisme qui permette de l'obtenir. Pour obtenir les informations de charge du site (comme la longueur de la queue d'attente pour la CPU, par exemple), il faut lancer un processus dont la tâche est d'échantillonner les données utilisées. Plus la période d'échantillonnage sera courte plus elle sera précise, en contre-partie elle sera pénalisante pour les performances du système. De plus, pour que cette information soit représentative de la charge future du site et pour éviter de prendre en compte des pics de charge résultant du lancement de tâches très courtes, il peut être avantageux d'obtenir l'indicateur de charge par le calcul de la moyenne des informations de charge sur un ensemble de valeurs de l'échantillonnage.

[Stu88] utilise un processus spécialisé qui incrémente un compteur. Si la différence entre deux interrogations sur la valeur du compteur est importante cela signifie que le site est peu chargé, par contre si la différence est faible cela signifie que le site est chargé. Cette méthode a l'avantage de ne pas interrompre la CPU, mais elle ne prend pas en compte les processus bloqués en attente d'une entrée/sortie.

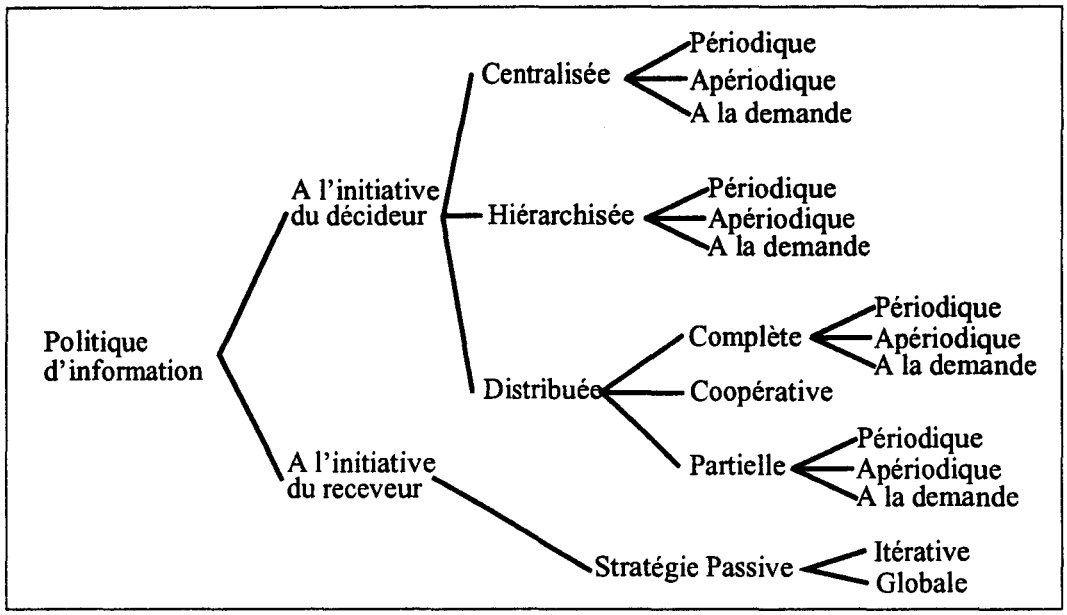


FIG. 1.13 - *Classification des mécanismes d'échanges d'informations*

1.6.2.2 Mécanismes d'échanges d'informations de charge entre noeuds

Les algorithmes de placement dynamique ont besoin en général pour fonctionner d'un état global de la machine. Dans le cas d'une architecture distribuée cet état global n'est pas disponible directement. Il faut mettre en place un mécanisme d'échange des indicateurs de charge locale à chaque site pour définir un état global de la machine.

Ce mécanisme d'échange garantit la validité de la vision que l'on a de l'état global de la machine. Si l'état global est régulièrement modifié et intégre parfaitement l'évolution de chaque site, alors l'algorithme de placement dynamique aura de bons résultats. Cependant, pour disposer d'un état global vraisemblable du système, il faudra échanger un grand nombre d'informations. Ces échanges représentent un coût pour le réseau de communication et un coût de traitement pour les sites qui vont devoir les traiter. Ceux-ci pénaliseront le traitement de l'application elle-même.

Différentes stratégies ont été développées, nous allons en présenter une classification (voir figure 1.13).

Remarque: Certains algorithmes de placement dynamique font un placement en *aveugle* c'est à dire qu'ils ne tiennent pas compte d'un quelconque indicateur de charge pour choisir le site destinataire d'une tâche. Dans ce cas aucun mécanisme d'échange d'informations n'est nécessaire.

Organisation des échanges centralisée

Dans ce type de stratégie d'échanges, il y a un noeud particulier *serveur* qui contient les informations de charge de l'ensemble des noeuds. Les échanges entre le *serveur* et les noeuds *clients* sont :

- soit à l'initiative des clients, ceux-ci envoient les informations de charges sans que le serveur n'ait besoin de faire la demande. Dans ce cas le client peut transmettre les informations de façon :

  - Périodique

    Le client déclenche un envoi de message vers le serveur tous les Δt. La détermination de cette valeur Δt va influencer directement sur la qualité de la valeur de l'état global du système, plus elle sera petite et plus l'état global du système sera juste, en contre-partie plus le réseau de communication sera encombré. Dans une variante de ce principe, le *serveur* pourrait prendre l'initiative de modifier la valeur de Δt dynamiquement en fonction des variations de l'indicateur de charge qu'il reçoit. Si cette variation est trop importante le serveur demande au client considéré de transmettre plus rapidement ses informations de charge.

  - Lors d'une modification importante de la valeur de la charge sur le noeud *client*.

    Cette fois le client n'envoie ses informations de charge que si une modification importante est intervenue depuis le dernier envoi [TL89].

- soit à l'initiative du *serveur*, qui demande périodiquement la charge à chacun des noeuds *clients* comme dans Remote Unix [LK87].

En général le *serveur* est placé sur un site dédié pour pénaliser le moins possible le reste de l'application. Ce type d'organisation est adapté aux machines ne comportant qu'un faible nombre de noeuds. Quand le nombre des noeuds devient important l'accès aux informations stockées sur le noeud *serveur* constitue un goulot d'étranglement par les noeuds *clients*, de même quand le nombre de demandes d'informations devient important. Elle a aussi le gros désavantage de ne pas être tolérante aux pannes, dans le cas d'un arrêt du noeud *serveur*, une duplication des informations sur n sites relais réduit les risques.

Une politique d'échanges centralisée peut conduire à une politique de choix d'un site qui peut être centralisée et dans ce cas c'est le même noeud *serveur* qui le prend en charge, ou bien elle est distribuée (voir paragraphe 1.6.2.4).

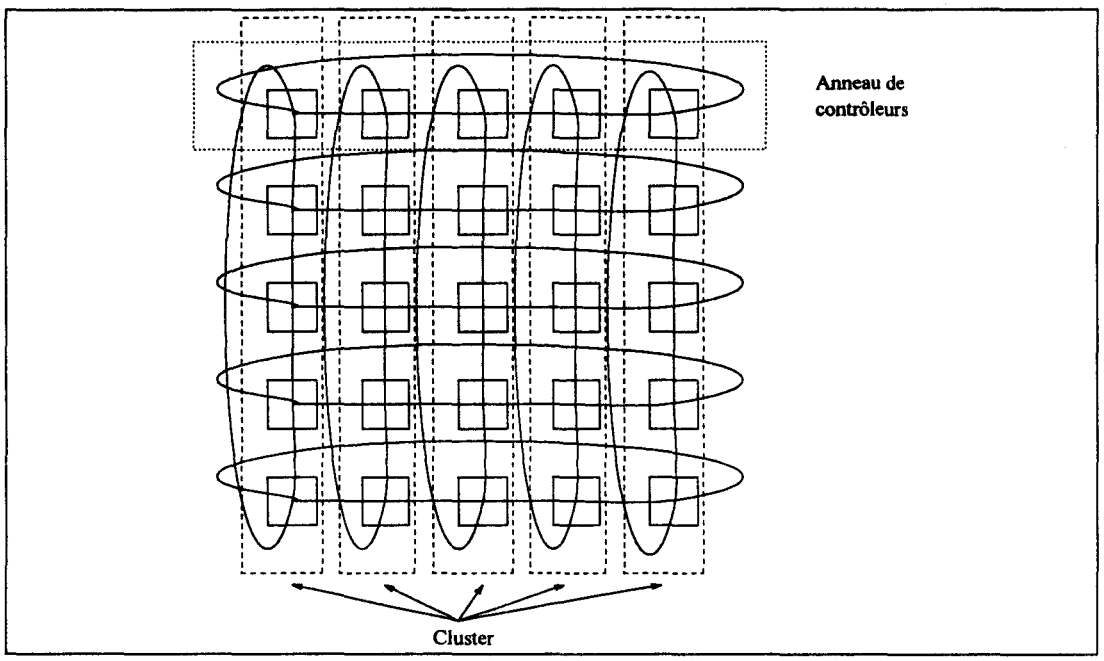


FIG. 1.14 - *Un exemple de topologie adapté à l'organisation des échanges hiérarchisée*

Organisation hiérarchisée des échanges

L'inconvénient majeur de l'organisation centralisée des échanges réside dans le fait qu'elle est difficilement extensible. Une réponse à cette critique est de définir des niveaux qui regroupent un sous-ensemble de sites de l'architecture. La détermination d'un niveau est directement dépendante de la topologie de la machine [Thi91]. Par exemple si la topologie représente une collection d'anneaux de site, chacun des anneaux reliant des noeuds. Un niveau sera l'ensemble des sites de l'anneau. A l'intérieur d'un niveau il y aura un mécanisme de collecte des informations de type centralisé vers un responsable de niveau Sa fonction sera de fournir un état de charge global du niveau aux autres niveaux de même rang. Cette structure peut se reproduire à l'infini ou presque. Ce genre de dispositif a l'avantage de réduire la distance des communications pour l'échange d'informations de charge et aussi de pouvoir mixer différentes méthodes d'échange des informations de charge (voir fig. 1.14).

Dans [RVV92] l'organisation des échanges est hiérarchisée (voir fig. 1.14) en deux niveaux. Dans un premier niveau, un contrôleur dispose des informations de charge d'un *Cluster* de sites organisés en anneau. Dans un second niveau les contrôleurs eux-mêmes organisés en anneau communiquent l'état de leur anneau par l'intermédiaire d'un jeton circulant. Chaque site peut lui-même être un ensemble de sites de niveaux inférieur (dans ce cas on à une organisation récursive), ou un processeur.

Organisation distribuée des échanges

Dans cette organisation chaque noeud dispose d'un état de la charge de la *totalité* des autres noeuds de la machine ou d'un *sous-ensemble.*

- La **connaissance complète** de la charge de chaque noeud nécessite un sur-coût non négligeable en terme de communication pour tenir à jour les informations. Différentes stratégies sont utilisées :

  - Diffusion périodique de la charge du noeud vers les autres [Zho88]
  - Diffusion de la charge du noeud vers les autres lorsqu'une modification importante a été constatée.

  Ces stratégies ne sont généralement utilisées que pour des machines disposant d'un nombre de noeuds assez faible, car le coût en communication est proportionnel au nombre de noeuds. Le temps de diffusion des informations devient trop grand pour assurer sa validité au moment de son utilisation par le site destinataire. Seules des adaptations sur l'architecture physique de la machine peuvent permettre de mettre en place ce genre de mécanisme d'échanges.

  La machine du projet ArMen [Das92] utilise une Couche Logique Reconfigurable (Voir fig. 1.15). Un site met à disposition de la CLR son information de charge. La CLR est chargée de fournir le numéro du site qui a la charge minimale et la valeur de cette charge minimale. L'introduction de la CLR permet de ne pas ajouter de sur-coût de communication pour cause de diffusion d'information de charge.

  Mis à part ces adaptations coûteuses et peu extensibles on préfère les stratégies qui ne disposent que d'une connaissance partielle de la charge des noeuds.

- Il existe de très nombreuses variantes des stratégies mettant en place une **connaissance partielle** de la charge des noeuds de la machine.

  - Dans [ELZ86, Zho88] les auteurs définissent une variable *MaxPoll* qui représente le nombre des machines qui vont être interrogées explicitement sur leur charge. Un tirage aléatoire est fait pour déterminer le nom des machines. Ces demandes explicites sont activées au moment du déclenchement de l'algorithme de choix du site destinataire. Cette méthode, qui est moins coûteuse qu'une diffusion globale (*MaxPoll* $< N$ Nombre total de sites), est cependant pénalisante par le fait qu'il faut, d'une part mettre en place un mécanisme de *Question-Réponse* qui double le nombre de messages pour obtenir la même information

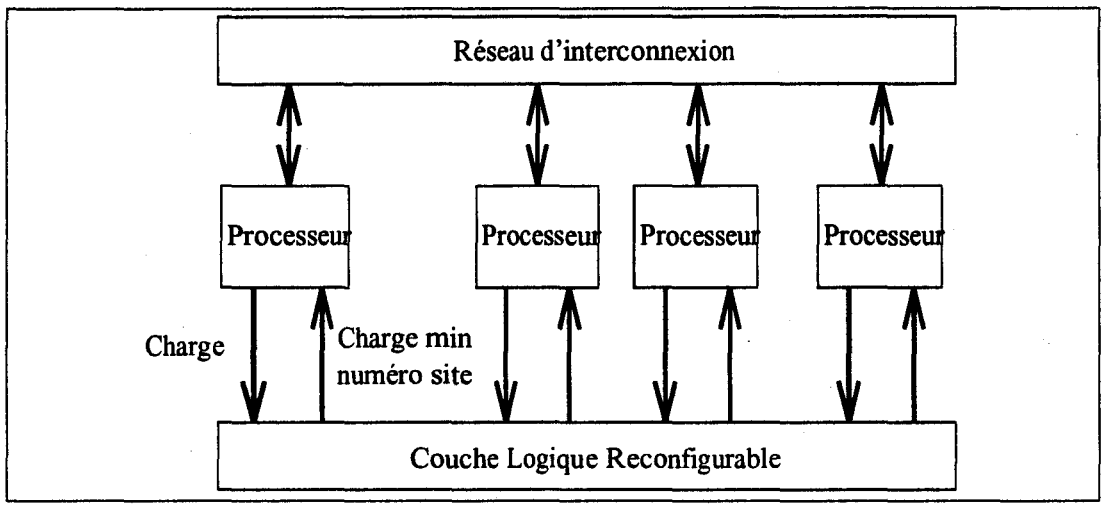


FIG. 1.15 - *L'utilisation d'une couche matériel pour l'aide à la diffusion des informations de charge*

sans demande explicite et que d'autre part elle ralentit le déclenchement de la création ou du déplacement de la tâche candidate au mouvement par l'attente des *Réponses*.

- dans [BS85] chaque noeud dispose d'un vecteur contenant la charge d'un sous-ensemble de noeuds, périodiquement le noeud modifie la valeur relative à son état dans le vecteur, et envoie la moitié des données du vecteur vers un noeud choisi par un tirage aléatoire. Le noeud destinataire met à jour ces données locales, il peut à son tour répéter le processus. L'état global de la machine est ainsi véhiculé au travers du réseau.

- Diffusion périodique de la charge d'un noeud vers les voisins immédiats.

  Le **jeton circulant**, les informations concernant la charge des sites sont rangées dans un vecteur. Le *Jeton* circule de site en site. Le site qui contient le *Jeton*, prend en compte les nouvelles valeurs de la charge des autres sites et modifie, si il y a lieu, la valeur correspondant à son indicateur de charge dans le vecteur, après quoi, il transmet le *Jeton* au site suivant. La méthode d'échange utilisant un *Jeton* est paramétrée par une fréquence de la circulation du jeton. Cette fréquence doit être adaptée à la variation de la charge des sites. Une fréquence trop petite réduirait la validité de l'information , une fréquence trop élevée augmenterait le sur-coût de communication inutilement. Une évolution dynamique de cette fréquence en fonction de la variation de la charge est donc souhaitable. Si entre deux passages successifs, le site constate une grande différence, il décide localement d'augmenter la fréquence. Le cas contraire est plus délicat, une diminution de la fréquence locale

peut induire une perte d'information sur les sites suivants. L'itinéraire du *Jeton* peut lui aussi être un paramètre, il est plus judicieux de choisir un itinéraire du *Jeton* qui suive un parcours le moins coûteux possible en terme de communication. On peut imaginer un itinéraire dynamique du *Jeton*, qui évolue pour fournir un état global de la machine plus rapidement aux sites qui en ont le plus besoin. La technique du *Jeton* circulant devient moins adaptée au fur et à mesure que le nombre de sites augmente, dans ce cas on peut mettre en place un mécanisme hiérarchisé de *Jetons*, chacun d'entre eux parcourant une partie de l'architecture, un *Jeton* particulier contenant l'information de charge de chacun des sous-parcours autonomes.

- Diffusion apériodique de la charge d'un noeud vers les voisins immédiats.

  Dans cette catégorie on trouve des exemples particuliers comme la méthode du **gradient** [LK87, PTS88, Tal91], où chaque noeud propage son état de charge de voisin en voisin. Cet algorithme de placement dynamique a un fonctionnement particulier qui ne rentre pas dans la décomposition donnée dans la figure 1.11, nous la développerons dans un paragraphe ultérieur, de même que la méthode des **"enchères"** [SS84, FYN88], où le noeud qui cherche à placer un processus lance une enchère vers un voisinage qui peut se limiter aux voisins immédiats, mais pourra s'étendre aux noeuds ayant une *distance* inférieur à D, avec D qui évolue dynamiquement en fonction de la charge du système.

- Diffusion de la charge par les messages échangés pendant l'exécution de l'application

  On utilise les messages échangés par les processus de l'application comme support pour échanger les informations de charge entre sites. L'information de charge est le plus généralement codée sur un entier long ou sur un flottant ce qui ne modifie pas de beaucoup la taille des messages échangés et par conséquent ne surcharge pas le coût de communication entre processus. Cette méthode nécessite un taux de communication important et largement réparti sur l'ensemble des sites de la machine. En effet, un site connaîtra uniquement la charge des sites avec lesquels il communique. Dans le cas où l'on ne dispose pas d'information on considère que la charge est faible, ce qui aura pour conséquence de favoriser le site dans le futur. L'inconvénient est que l'on ne dispose pas d'une information globale fiable, et que cette information dépend fortement de l'application qui s'exécute. Pour résoudre ce problème, on associe en général cette technique à une méthode d'échanges classique présentée précédemment pour rendre l'information plus fiable mais cette fois en utilisant des paramètres qui entraîneront un coût beaucoup moins important en terme de communication.

Dans les stratégies présentées l'initiative de l'échange est prise par le noeud qui doit faire le choix du noeud destinataire du processus, elles sont appelées **stratégies actives**, il existe des **stratégies passives**, où ce sont les noeuds inactifs qui demandent du travail aux noeuds qui ont une charge importante [KR91, Bla92]. Les stratégies passives sont plus largement utilisées, quand le système gère la migration de processus (voir paragraphe 1.6.2.3).

Les stratégies passives d'échanges d'informations

Dans les stratégies passives ce sont les sites inactifs qui demandent du travail à leur voisinage. Un site est considéré comme inactif lors qu'il n'a plus aucune chance d'avoir du travail sans en demander. Le voisinage correspond à l'ensemble des sites qui vont être interrogés, qui correspond en général aux sites les plus proches physiquement. Il y a deux méthodes pour traiter les requêtes émanant de sites inactifs.

- Soit le site qui reçoit une demande de travail, mémorise l'information pour lui transmettre du travail quand il en aura trop. Dans ce cas le site inactif doit demander du travail à l'ensemble du voisinage sans attendre de réponse immédiate. Sinon il ne peut pas déterminer quand il doit faire une demande à un autre site du voisinage. Le désavantage de cette technique est que le site inactif risque d'être submergé de travail si l'ensemble du voisinage lui envoie une partie de leur travail.

- Soit le site inactif questionne un à un les sites du voisinage. Si le site interrogé répond qu'il n'a pas de travail en trop, alors le site inactif passe au site suivant. Si l'ensemble du voisinage répond par la négative, alors le site s'endort pendant une certaine période et recommence l'exploration. Si le site interrogé a du travail alors le site inactif le décharge et il arrête la recherche de travail.

Il existe de nombreuses possibilités sur le choix et le volume des tâches qui vont être transférées. Dans [KW91] les auteurs donnent une étude comparative de différentes approches de répartition du travail.

L'avantage des stratégies passives est qu'elles surchargent très peu la machine lorsque les sites ont du travail, et que la surcharge est répartie sur les sites qui n'ont pas de travail. Seules les périodes de lancement d'une application ou de terminaison d'une application entraîne un délai dans le traitement des nombreuses requêtes.

Présentation de la méthode du GRADIENT

La méthode du *Gradient* développée ici est reprise de [LK87]. L'indicateur de charge n'est plus une valeur, mais un niveau de charge: *Libre, Actif, Chargé.* L'indicateur passe d'un niveau à un autre en fonction de deux seuils: *Seuil_Bas, Seuil_Haut.* Pour une description de l'évolution d'un niveau à autre voir 1.6.2.1.

Le niveau de charge conditionne son comportement vis à vis des entités à créer.

1. *Libre:* Les demandes de création locale sont satisfaites localement. Les propositions émanant de sites distants sont acceptées.
2. *Actif:* Les demandes de création locale sont satisfaites localement. Les propositions des sites distants sont refusées.
3. *Chargé:* Le site cherche un site distant pour satisfaire toutes demandes de création locale.

Les valeurs données aux seuils *Seuil_Bas, Seuil_Haut*, définissent la politique de transfert. En effet si *Seuil_Bas* = *Seuil_Haut* = 0 alors le site cherche à placer les entités sur les autres sites (répartiteur de processus dans le réseau). Si *Seuil_Bas* = *Seuil_Haut* = ∞ alors toutes demandes de création locale et distante sont acceptées (serveur de processus). Enfin si *Seuil_Bas* = 0 *et Seuil_Haut* = ∞ le site n'accepte aucun processus de l'extérieur mais n'en donne pas aux autres sites (ne participe pas à la stratégie de placement dynamique) [BSS91].

Un site i accepte une demande de création distante lorsque son niveau de charge est *Libre*, ou encore si la porte $g_i = 0$ (ouverte) avec:

$$\begin{cases} g_i = 0 & \text{si } etat(i) = Libre \\ g_i = D + 1 & \text{sinon, avec } D \text{ le diamètre de la topologie de l'architecture} \end{cases}$$

Les auteurs définissent la notion de surface de proximité, qui indique pour chaque site la distance[1] qui le sépare d'un site libre.

$$\begin{cases} prox_i = \min\{d_{i,j},\ avec\ g_j = 0\} & \\ prox_i = D + 1 & \text{si il n'existe pas de site j tel que } g_j = 0 \end{cases}$$

Dans la figure 1.16 les sites 5 et 13 sont dans un niveau de charge *Libre*, ce qui donne la surface de proximité reproduite.

Cependant un site ne peut pas calculer la surface de proximité à un instant donné. Les auteurs introduisent la notion de proximité propagée:

1. la distance $d_{i,j}$ est égale au nombre de liens qui séparent les sites i et j

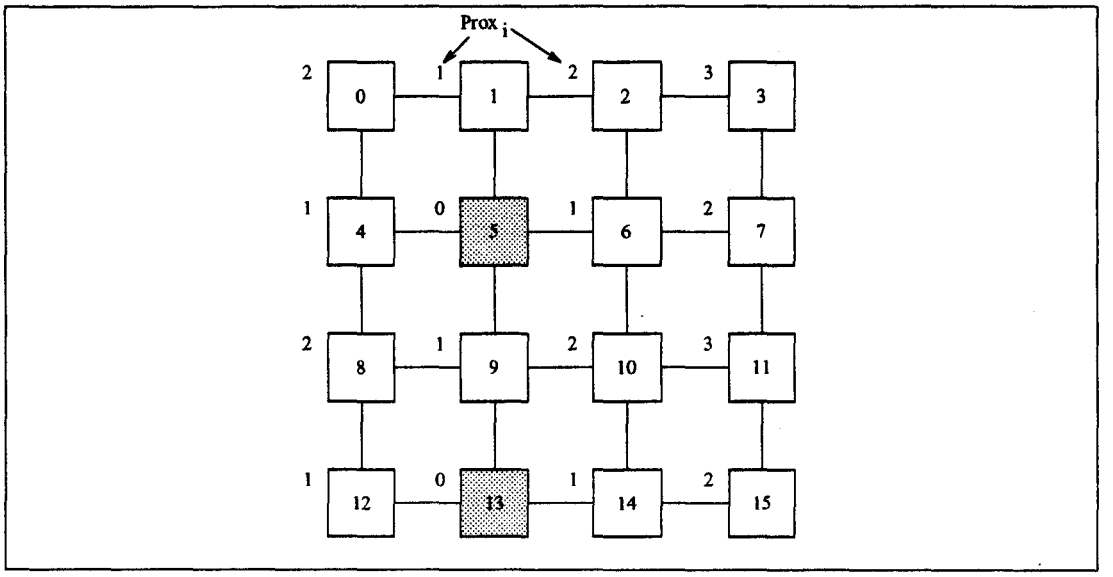


FIG. 1.16 - *Surface de proximité*

$$pp_i = \min\{g_i, \min\{pp_j,\ avec\ d_{i,j} = 1\} + 1\}$$

et montrent que si le système est stable (pas de variation de charge) alors on a : $pp_i = prox_i$.

A chaque modification du niveau de charge le site en informe les sites voisins comme présenté par l'algorithme de la figure 1.17.

La fonction *Diffuse_nouveau_pp()* envoie la nouvelle valeur de pp_i aux voisins pour assurer la propagation de leur proximité.

L'équilibrage de charge se fait d'un site ayant un niveau de charge *Chargé* vers un site ayant un niveau de charge *Libre*. Le transfert se fait par migration de la tâche. Le chemin emprunté est determiné au niveau de chaque site parcouru. Le site voisin utilisé pour transférer la tâche est celui ayant une valeur pp_j minimum. Cette technique est utilisée jusqu'à ce que la tâche arrive sur le site avec une valeur de $pp_i = 0$. Dans le cas où la surface de proximité est dans un état instable, l'algorithme de transfert d'une tâche peut amener à des situations de bouclage, c'est à dire que la tâche est transférée de site en site indéfiniment, pour éviter ce genre de mésaventure, lors d'un transfert, on limite le nombre de sites parcourus.

Présentation de la méthode par enchères

L'algorithme des *enchères* se décompose en 4 composantes

1° Opportunité de lancer des enchères;

```
proc calcul_pp(i)
    Old_pp_i = 0
    tq (1) faire
        Etat ← Valeur_Etat()
        choix Etat faire

            cas Libre :
                pp_i ← 0
            cas Actif :
                pp_i ← 1 + min{pp_j} avec d_{i,j} = 1
                si pp_i > D + 1 alors pp_i = D + 1 fsi
            cas Chargé :
                pp_i ← 1 + min{pp_j} avec d_{i,j} = 1
                si pp_i > D + 1 alors pp_i = D + 1
                sinon
                        si min(pp_j) < pp_i alors
                            Tranfert d'un tâche sur le site j avec pp_j minimum
                        fsi
                fsi
        fchoix
        si Old_pp_i ≠ pp_i alors
            Diffuse_nouveau_pp()
        fsi
    fait
```

FIG. 1.17 - *Algorithme de calcul et de diffusion de la valeur de proximité*

Pour cela l'algorithme a besoin de connaître les caractéristiques des tâches locales (état, coût d'exécution total, coût d'exécution déjà consommé, etc ...), il a besoin de connaître les informations de charge du site (mémoire disponible, puissance, coût de communication avec les voisins, ...). L'ensemble de ces informations est utilisé comme base de faits dans un système expert qui renvoie un entier par tâche sur le site. Si la valeur de cet entier est nulle la tâche n'est pas transférée. Plus la valeur est petite tout en étant différente de zéro et plus la tâche a des chances d'être transférée. Un seuil définit la valeur maximale au dessus de laquelle la tâche n'est plus transférable.

A la suite de cette selection, il reste une liste de tâches (éventuellement vide) pouvant être candidates au transfert vers un autre site.

2° Transmission des requêtes d'enchères;

Dans le cas où la liste obtenue n'est pas vide, à cette étape on détermine l'ensemble des tâches pour lesquelles on va lancer des enchères. Dans le cas où l'on décide de lancer des enchères, celles-ci sont envoyées vers un ensemble de sites. Cet ensemble est défini par un paramètre **distance** qui correspond au nombre de lieux possibles entre le site émetteur de l'enchère et le site récepteur. Cette distance varie dynamiquement en fonction de la charge globale de la machine.

3° Traitement des requêtes d'enchères;

Dans les requêtes d'enchères on retrouve les caractéristiques de la tâche qui fait l'objet de l'enchère. Le site qui reçoit la requête, la traite comme une tâche locale et l'introduit dans la base de fait de son système expert, en sortie il obtient un entier qui correspond à la proposition du site en réponse à la requête de l'enchère. Si cette réponse est supérieure au seuil défini par le site, alors celui-ci ne participe pas à la requête d'enchère.

4° Transfert des tâches;

Le site qui a lancé une requête d'enchère attend les réponses pendant un certain temps Δt qui est fonction du paramètre distance. Chaque offre reçue est ajustée en fonction de la distance qui sépare les deux sites (émetteur de la requête et émetteur de l'offre). Après ajustement la meilleure offre est choisie, c'est à dire que la tâche est transférée vers le site ayant émis la meilleure offre. Si aucune offre n'est valide c'est à dire que toutes les offres sont moins bonnes que l'offre locale alors on relance une requête d'enchère en ajustant le paramètre distance.

Contrairement aux autres techniques l'algorithme des enchères nécessite de connaître les caractéristiques de la tâche pour pouvoir la placer. Le choix d'un site pour chaque demande de transfert d'une tâche entraine un trafic non négligeable

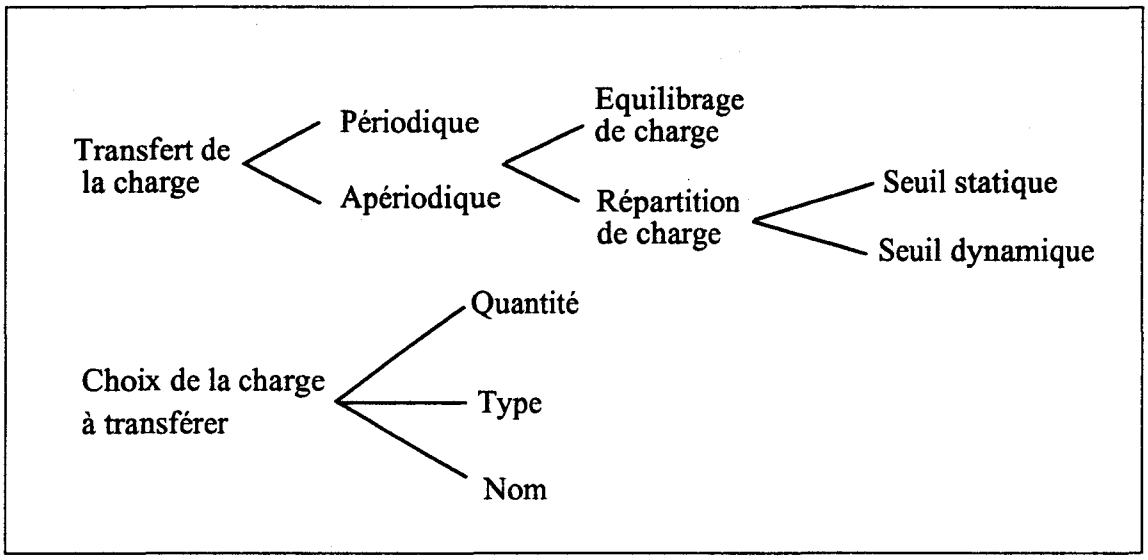


FIG. 1.18 - *Présentation des mécanismes de transfert*

de communication. Ce type d'algorithme est donc destiné à être utilisé par des applications à gros grain et ne nécessitant pas souvent une remise en cause de la répartition des charges. Cette technique est comparable à l'approche micro-économique proposée dans [FYN88]. Un processeur est vu comme un prestataire de service et les tâches comme les demandeurs de service. Une tâche dispose d'un capital de base qu'il va dépenser en demandant des services aux processeurs. Les tarifs pratiqués par les processeurs dépendent de la loi de l'offre et de la demande, et sont diffusés sur l'ensemble du réseau pour être connus par l'ensemble des tâches.

1.6.2.3 Choix de la fréquence des transferts et Choix des processus à transférer

Dans les algorithmes de placement dynamique on distingue deux types d'objectifs, ceux qui font de la répartition de charge, et ceux qui font de l'équilibrage de charge.

- Dans le cas d'un **équilibrage de charge** (*Load Balancing*), l'algorithme doit garantir que la différence de charge entre deux noeuds de la machine soit minime. A chaque opération pouvant entraîner une modification de la charge d'un noeud, comme la création d'un nouveau processus, ou la destruction d'un processus, il faut déclencher l'algorithme de recherche des processus transférables.
- Pour la **répartition de charge** (*Load Sharing*), l'algorithme a pour objectif de garantir que la charge d'un site ne dépasse pas un certain seuil *ValSeuil*,

alors que d'autres noeuds de la machine sont inactifs. L'algorithme de recherche des processus transférables est déclenché quand $IC(i) \geq ValSeuil$, avec $IC(i)$ qui représente la valeur de l'indicateur de charge du noeud i.

La mise en place d'un algorithme qui fait de l'équilibrage de charge donne de meilleurs résultats qu'un algorithme de répartition en terme d'utilisation des ressources, cependant le coût de la mise en place d'un tel algorithme relativise les gains en terme de temps d'exécution[KL87, KL88b]. La valeur *ValSeuil* est définie statiquement au lancement du système ou évolue dynamiquement en fonction de la charge globale du système, elle ne doit pas être trop faible pour garantir un gain en temps d'exécution par rapport au coût de transfert d'un processus, ni trop élevée pour garantir un minimum de répartition. Dans [Cho90], le déclenchement de l'algorithme de transfert est décrit de la manière suivante :

$n = IC(du\ noeud\ courant)$
si $n > 0 \wedge \exists\ i / IC(i) < f(n)$ **alors**
exécuter le processus sur le noeud i
sinon
exécuter le processus localement
fsi

Pour que l'algorithme fonctionne convenablement, il faut choisir $f(n)$ tel que $f(n) < n$. La valeur de $f(n)$ peut prendre les valeurs $n - 1$, n *div* 2, n *div* 3.

La définition des processus transférables peut être différente d'un système à l'autre. Dans certains systèmes les processus qui sont candidats au transfert sont uniquement ceux que l'on veut créer dynamiquement(algorithme de répartition *non-préemptif*), leur localisation ne sera pas remise en cause au cours de leur exécution. Dans d'autres systèmes les processus candidats au transfert peuvent être des processus en cours d'exécution (algorithme de répartition *préemptif*), dans ce cas un mécanisme de **migration** permet de déplacer le processus dans un état cohérent d'un noeud vers un autre.

De plus certains algorithmes de placement dynamique, le processus candidat au transfert doit avoir certaines propriétés. Ces propriétés garantissent que l'examen de son transfert sera rentable en terme de gain d'exécution ou possible dans le cas d'une architecture hétérogène. Comme exemple de propriétés on peut citer le temps de consommation CPU déjà utilisé, en effet on a constaté que plus un processus à consommer du temps CPU plus il en consommera dans le futur, il peut donc être intéressant d'envisager son transfert.

La fréquence de déclenchement dépend des possibilités du système. c'est à dire si il dispose de la possibilité de migration d'un processus au cours de son exécution et de la politique de répartition que l'on veut mettre en place. Dans le tableau 1.1 sont définis les événements qui déclenchent la recherche des tâches à

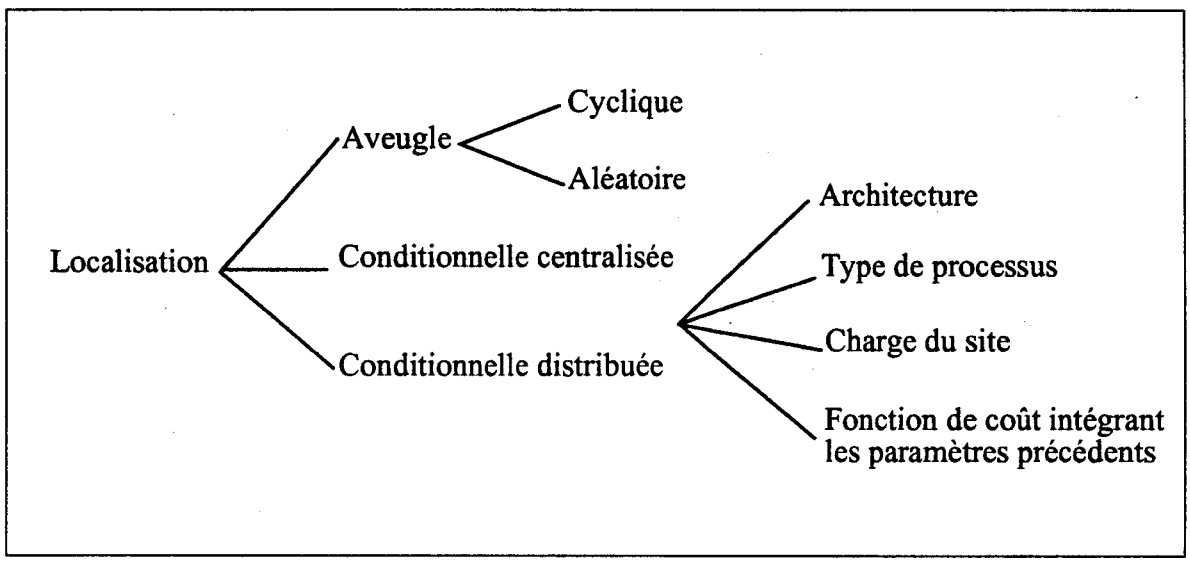


FIG. 1.19 - *Présentation des mécanismes de choix du noeud*

transférer.

Possibilités	partage	équilibrage
migration	– en fonction du seuil *ValSeuil*	à chaque création à chaque destruction d'un processus
sans migration	– à chaque création et en fonction du seuil *ValSeuil*	à chaque création

TAB. 1.1 - *Fréquence de déclenchement de l'algorithme de transfert en fonction des possibilités du système et du type de répartition mis en place*

1.6.2.4 Choix du noeud destinataire

Si l'on suppose que l'on dispose d'une valeur *étalon* permettant de comparer la charge de chacun des sites de la machine, et d'un dispositif permettant de disposer d'un état global de la machine, on est capable de mettre en place un dispositif de choix d'un noeud destinataire à chaque fois qu'est déclenchée une demande de création et/ou de transfert d'une tâche. Comme pour le mécanisme d'échanges d'informations, il y a deux techniques possibles : centralisée ou distribuée.

Choix du noeud sur un site unique

Comme dans le cas d'une stratégie d'échanges centralisée, un noeud *serveur* qui contient les informations de l'ensemble des noeuds est dédié à la détermination du meilleur noeud pour une tâche. Cette solution a les mêmes inconvénients que ceux cités pour la stratégie d'échanges centralisée : goulot d'étranglement quand le nombre de demandes est important et une mauvaise tolérance aux pannes.

Choix du noeud décentralisée

Dans ce cas il faut disposer d'un état global ou partiel de la machine qui permette au noeud de choisir localement le meilleur site pour la tâche. Un état global fiable est plus difficile à obtenir ce qui a pour conséquence de rendre l'algorithme de décision moins précis.

Technique de choix du meilleur noeud

Le choix du noeud destinataire est généralement le noeud qui est connu par l'algorithme de décision comme ayant la plus petite valeur d'indicateur de charge, ce qui se traduit par le choix du noeud qui a le moins de processus présents, le moins de processus prêts, ou le moins de messages en attente. Ce choix étant lié à la pertinence de l'état global du système certains mécanismes choisissent le premier noeud acceptable plutôt que le meilleur à tout prix.

La valeur de l'indicateur peut être un paramètre d'une formule $f(x_1, x_2, \ldots, x_n)$ incluant d'autres paramètres comme la distance dans l'algorithme basé sur les "enchères", où le noeud destinataire retenu est celui qui a donné la meilleure enchère et qui est le moins loin du noeud origine de l'enchère, mais aussi la date de la dernière modification de l'indicateur de charge ce qui permet d'introduire un coefficient de vraisemblance de l'état global de la machine.

Dans ce qui précéde, le choix du noeud pour la tâche repose uniquement sur l'état global de la machine sans tenir compte de la tâche à placer. Dans [Fol93] l'auteur introduit différents *critères d'allocation* pour un placement en fonction du *comportement des applications* :

- le temps processeur de chaque programme;
- la mémoire nécessaire à l'exécution de chaque programme;
- les fichiers utilisés et créés par chaque programme;
- les communications entre les programmes;
- tout autre critère, en fonction d'algorithmes écrits par les programmeurs.

Par cette technique on essaye d'adapter l'algorithme de choix d'un meilleur noeud en fonction du programme à placer. Cela nécessite de la part du programmeur un travail de *déclarations d'intention* définissant explicitement les valeurs.

Dans les travaux que nous allons présenter dans la suite du rapport, nous essayons de définir une classification des tâches suivant la nature des instructions qu'elles contiennent. Cette classification permettra de proposer un algorithme particulier par classe de tâches.

Choix du noeud en aveugle

Dans certains algorithmes [ZF87, ELZ86, Ath87], le placement ne repose sur aucune information d'état global du système. Dans ce cas le noeud qui va accueillir la tâche est déterminé de manière aléatoire, chaque noeud ayant la même probabilité d'être choisi, ou de manière cyclique [WM85] chaque noeud connait le numéro du noeud qui a été choisi lors du dernier envoi. Cette technique qui donne des meilleurs résultats qu'une exécution sans aucun placement dynamique, a l'avantage de ne pas ajouter de sur-coût pour sa mise en place. Cependant elle peut être dangereuse lorsque le noeud choisi est déja surchargé.

Certaines variantes se distinguent par leur manière de gérer le comportement du noeud destinataire vis à vis du transfert d'une tâche et des conséquences sur le noeud émetteur.

- Possibilité de refus par le noeud destinataire [ST85];

  Le noeud destinataire a la possibilité de refuser une tâche dans ce cas la tâche est soit renvoyée vers l'emetteur qui l'exécute localement, soit le noeud transfert la tâche vers un autre noeud. Cette dernière possibilité peut engendrer des instabilités lorsque le système est globalement surchargé, on limite dans ce cas le nombre de transferts possibles pour une tâche.

- Mémorisation des événements [SK90];

  Comme précédemment le noeud destinataire peut refuser un transfert. Chaque résultat d'un transfert est mémorisé. Dans le cas d'un refus du noeud destinataire la probabilité que ce noeud soit à nouveau choisi est diminué, dans le cas d'une acceptation la probabilité augmente.

- Modification des probabilités adaptées au noeud [CK79];

  Contrairement à un choix aléatoire simple, dans ce type d'algorithme les probabilités P_{ij} sur un noeud N_i sont adaptés à chaque noeud N_j possible. Les P_{ij} peuvent être arbitraires ou en fonction de l'éloignement des noeuds N_i et N_j, ou encore en fonction du type de noeud pour une architecture hétérogène.

Le choix en aveugle du noeud destinataire donne en général de bons résultats lorsque les tâches qui sont à placer sont d'un grain fin et pour les applications qui déclenchent une activité régulière, comme c'est le cas dans les applications numériques, dans ce cas un placement cyclique est bien adapté.

1.6.2.5 Conclusion sur le placement dynamique

L'utilisation du placement dynamique apparaît lorsque l'on veut permettre une adaptation de l'exécution d'une application ou d'une collection d'applications à la configuration rencontrée au lancement de l'application sur la machine. Le placement dynamique intervient aussi lorsque l'application a une activité qui évolue dynamiquement et de manière imprévisible statiquement.

Nous avons identifié quatre étapes importantes dans les algorithmes de placement dynamique qui sont plus ou moins liées entre elles et qui doivent être adaptées à l'architecture de la machine visée.

1° Le calcul de l'indicateur de charge

L'indicateur de charge doit avoir un certain nombre de propriétés dont une, essentielle qui consiste à déterminer l'état de charge d'un noeud dans un futur proche. Il doit être adapté à la machine et au type d'applications qui vont y être exécutées. Sur les systèmes distribués faiblement couplés, les tâches sont généralement d'un grain moyen à gros. On parle alors plutôt en terme de taux d'utilisation de la CPU, et l'indicateur de charge exprime alors un niveau de charge. Dans les machines parallèles, les tâches ont un grain fin , dans ce cas un compteur de tâches est généralement plus adapté.

2° La stratégie d'échanges d'informations

Ce mécanisme qui permet de disposer d'un état global ou partiel de la machine est celui qui entraine le plus grand sur-coût de charge. Différentes stratégies s'affrontent pour s'adapter au type d'architecture utilisé.

Le cas d'un serveur dédié qui centralise l'état de chaque noeud est facile à mettre en oeuvre mais il est destiné aux machines constituées d'un nombre limité de noeuds donc difficilement extensible, de plus il a l'inconvénient d'être peu tolérant aux pannes. Les solutions décentralisées sont plus extensibles cependant les coûts de communication rendent l'information peu fiable ou surchargent le système. Les solutions hybrides où l'état global est géré par un sous ensemble de noeuds ou bien une vision sur les noeuds d'un état partiel de la machine deviennent des solutions intéressantes lorsque le nombre de noeuds devient important.

3° La fréquence des transferts et le choix des tâches à transférer

Nous avons vu que la fréquence des transferts dépend de la politique de répartition qui est mise en place et les possibilités du système. Pour les politiques de répartition, il existe deux approches : l'équilibrage de charge et le partage de charge. Dans le cas du partage de charge, on envisage le transfert quand la charge locale dépasse une valeur seuil. Pour l'équilibrage de charge, le transfert est envisagé à chaque modification de la charge. Le choix des tâches à transférer doit garantir qu'un transfert va apporter un gain en terme d'exécution ou en terme d'utilisation des ressources. Dans les systèmes distribués qui manipulent des tâches d'un grain important, la migration est généralement envisagé. Pour les architectures parallèles qui manipulent des tâches d'un grain fin, la migration est difficilement rentable, sauf pour les applications très irrégulières, dans ce cas les possibilités de migration permettrait de faciliter le travail du répartiteur.

4° Le choix du noeud destinataire

Dans les architectures qui n'ont pas de notion de distance entre noeuds, c'est à dire que chaque noeud est voisin de l'ensemble des autres noeuds, le choix se résume à trouver le noeud avec l'indicateur de charge le plus faible. Dans les machines parallèles avec une topologie particulière, les tâches tirent avantage a être placées en fonction de l'indicateur de charge mais aussi en fonction de la distance. Cela pour limiter les coûts de communication qui risquent d'être importants si elles communiquent beaucoup.

Le choix du noeud en aveugle peut se révéler être une bonne solution lorsque les tâches sont de grain fin et que l'application décrit un schéma régulier d'exécution.

Plusieurs classes d'algorithmes de placement dynamique existent. Chacune répond à une exigence particulière qui est liée principalement à l'architecture. Depuis quelques temps des recherches s'orientent vers des algorithmes qui introduisent la notion de caractéristiques d'une tâche pour pouvoir la placer plus finement et augmenter ainsi les gains en terme d'exécution ou en terme d'utilisation des ressources [Fol93].

1.7 conclusion

Comme on le voit trop souvent dans la littérature, le placement d'une application ne se résume pas à l'utilisation d'un des deux modèles de placement : le placement statique, ou le placement dynamique.

Le placement consiste d'abord à savoir et à connaitre l'entité que l'on veut

placer. Dans les applications du type bases de données réparties, on cherche à placer des données qui sont stockées sous forme de fichier. Dans les applications de type calcul numérique on cherche à placer des structures de données régulières comme une matrice par exemple. Dans les applications qui consistent à rechercher la meilleure solution dans un espace qui évolue dynamiquement, on cherche à placer la structure irrégulière engendrée. Dans les applications manipulant peu de données mais qui entrainent un traitement important, soit on connait statiquement l'ensemble des traitements et on cherche une solution au placement des traitements statiquement, soit on ne le connaît pas et les traitements sont placés dynamiquement au cours de l'exécution de l'application. L'inventaire de ces différents cas montre bien l'importance de la question suivante:

- Quelle est l'entité à placer?

Le type de l'architecture influence le placement. Dans les architectures SIMD, on va s'intéresser au placement des données, le traitement étant le même pour tous les noeuds. Dans les architectures MIMD, on va essayer au maximum de placer les données et le code qui va les traiter sur des noeuds voisins, si le coût de communication est important ou constitue un goulot d'étranglement, sinon on va placer les données ou le code en fonction de l'importance de l'un ou de l'autre pour l'application. Le grain des tâches manipulées par la machine et le nombre de noeuds sont des données qui modifient le traitement du placement d'une application. Ce qui conduit à se poser la question suivante:

- Quelle est l'architecture de la machine parallèle?

Les objectifs d'un algorithme de placement sont en général d'obtenir une configuration qui conduise à l'utilisation optimale de la machine parallèle. Cette notion d'optimale peut se comprendre en terme d'utilisation de ressource ou bien en terme de temps minimum d'exécution pour les applications temps réels en particulier. Un certain nombre de fonctions de coût permettent de préciser les objectifs d'un algorithme de placement. L'objectif peut traduire la volonté d'un certain niveau de tolérance aux pannes. Là encore, une formulation sous forme d'une fonction de coût et une liste de contraintes déterminent un placement particulier.

- Quel est l'objectif du placement?

Le placement d'une application quelque soit le modèle utilisé ou les objectifs définis ne donnera un bon résultat que si à chaque niveau de développement de l'application on aura aussi pensé à son placement. Nous avons identifié les niveaux suivants:

- les directives de l'utilisateur;

- le découpage de l'application;
- l'analyse du source de l'application;
- le chargement;
- et enfin le placement dynamique quand il est nécessaire.

Le problème de l'intégration des différentes informations obtenues à chaque niveau est un problème ouvert. En effet quand les informations sont contradictoires, quelle est la décision à prendre?

Ce chapitre se termine par la présentation d'une synthèse des travaux réalisés dans le domaine du placement statique et plus particulièrement dans le domaine dynamique. A cela plusieurs raisons, aujourd'hui les machines parallèles sont de plus en plus vues comme un système distribué incluant un ensemble de ressources, représentées par une collection de noeuds hétérogènes et partagées par plusieurs utilisateurs. Ce type d'architecture est mal traité par l'ensemble des algorithmes de placement statique. De plus notre étude se situe dans un contexte objet (voir chapitre II) dont le modèle d'exécution est basé sur un mécanisme de processus communicants créés dynamiquement, là encore un cas de figure mal traité par le placement statique.

Chapitre 2

Présentation du projet PVC

2.1 Introduction

Le projet PVC (Processeur Virtuel de Classe) a pour but de faciliter l'utilisation des nouvelles architectures de machines composées de plusieurs processeurs en proposant des outils qui rendent leur usage plus aisé et plus efficace. Pour cela nous utilisons le concept de programmation basé sur l'approche objet actif qui permet une programmation modulaire montrant de grandes qualités au regard du génie logiciel [Mey88], tout en donnant la possibilité d'exploiter les architectures parallèles.

Les machines cibles

Les machines de type multicomputer (voir chapitre 1) qui intéressent le projet PVC se caractérisent par l'absence d'une mémoire physique commune, un mode de contrôle d'exécution MIMD. Elles se composent d'un ensemble de noeuds regroupant un processeur, une mémoire, une unité de communication et d'un réseau de communication rapide reliant les noeuds. Dans cette catégorie de machines on intégrera les réseaux de stations de travail qui constituent un cas particulier avec des noeuds très puissants et un réseau de communication relativement lent par rapport à ceux rencontrés dans les architectures de machines massivement parallèles.

Les langages parallèles orientés objets

Les langages parallèles orientés objets sont reconnus comme étant bien adapté à la programmation de machines multicomputer [Mey88, YT87]. Il existe plusieurs types de langages parallèles orientés objets, qui se différencient par la façon d'introduire le parallélisme (ou la notion de processus) dans le monde des objets [Cou92]. Dans certains cas :

- Le nombre de processus peut être inferieur ou égal au nombre d'objets

  C'est le programmeur qui spécifie les objets de son application qui représenteront un processus (Concurent C++). D'autres langages intégrent une classe particulière qui représente les processus dans le monde des objets (langage C++, Presto, Smalltalk80, Trellis/Owl), chaque instance de cette classe (ou des classes qui l'ont comme ancêtre) crée un nouveau processus. La notion d'activité est aussi introduite dans la syntaxe de certains langages (Emerald) en ajoutant une *Process section* qui va être exécutée par un nouveau processus. Les langages à acteurs [Agh86] attachent implicitement la notion de processus à la notion d'objet. les acteurs communiquent par envoi de messages de type requête, ou réponse. Ce qui permet de garantir l'encapsulation des données dans l'objet. L'acteur exécute la partie de son code (*le script*) en fonction du message qu'il a reçu.

- Le nombre de processus peut être supérieur au nombre d'objets

  Pour augmenter le degré de parallélisme, on introduit la possibilité pour l'objet d'exécuter plusieurs méthodes en même temps (parallélisme intra-objet). Ce type de langage (PO, Guide, Dragoon, Act++) introduit un mécanisme de synchronisation permettant de garantir l'intégrité des données.

- Fragmentation des objets

  La représentation des objets sur la machine peut permettre d'introduire encore du parallélisme. Dans le cas où l'objet, dans sa totalité, c'est à dire les données et les méthodes est localisé sur un même site, l'exécution simultanée de plusieurs méthodes engendrera un pseudo-parallélisme. Par contre si l'objet est fragmenté (éclaté) sur plusieurs processeurs, on obtient un vrai parallélisme, tout en permettant une répartition plus lissée de la charge sur les processeurs [SCM+91, BB91]. Cette fragmentation n'est pas sans inconvenient. En effet, lors de la fragmentation d'un objet, il est important de continuer à garantir l'encapsulation des données, et permettre l'exécution de toutes les méthodes sur l'ensemble des données de l'objet. Il faut donc mettre en place un mécanisme qui va permettre la localité du code et des

données sur un même site soit par :

- La migration du code des méthodes,

  Le code de la méthode est transporté par le réseau de communication vers le processeur contenant les données, pour être exécuté. Le système *Emerald* [JLHB88] utilise cette méthode en cas de défaut de code.

- La migration des données

  Cette fois ce sont les données qui sont transmises par le réseau de communication vers le site contenant les méthodes [SCM+91].

- La duplication du code ou/et des méthodes

  La duplication sur plusieurs processeurs peut être un palliatif au déclenchement trop fréquent de méthodes entraînant un défaut de localité, mais a pour inconvénient de surcharger la mémoire des processeurs [Ben87]. La duplication des données est plus délicate, car elle nécessite la mise à jour des différentes copies des données après une modification [SB88].

Dans ce chapitre , après avoir montré comment le projet PVC apporte sa contribution à la conception et l'implantation de langages parallèles orientés objets sur machine du type multicomputer, nous allons présenter comment est introduit un mécanisme de répartition du code et des processus dans la plateforme de développement PVC. Puis, nous donnerons une implantation dans le run-time qui a été développé.

2.2 L'apport du projet PVC à la conception et l'implantation de langages parallèles orientés objets sur multicomputer

2.2.1 Un support d'exécution de l'environnement PVC

Afin de garantir une indépendance de l'environnement PVC vis à vis de l'architecture rencontrée, Luc Courtrai [Cou92] a introduit la notion de **Composant Actif de communication** (Cac) qui permet un découpage fin en terme d'activités concurrentes.

2.2.1.1 Le Composant Actif de Communication

Le Cac est l'entité active unique qui va être manipulée par l'environnement PVC. Un Cac regroupe (voir la Fig. 2.1) :

- Un **processus léger**; c'est à dire un *thread* dans la terminologie courante [Sun88, Per89]. Celui-ci va exécuter une partie de code encore appelée le *comportement* du composant. Le comportement dans la pratique et pour fixer les idées, correspondra au code d'une méthode dans un objet, un fragment d'objet, ou bien encore un objet.

- Un **environnement local**; qui constitue les données privées du composant. Il peut contenir des variables, des fonctions, ou encore des références vers d'autres composants. Ces données ne sont visibles que de lui seul.

- Une **boîte aux lettres**; qui stocke tous les messages destinés au composant. L'adresse de cette boîte aux lettres identifie le composant dans le système de communication de l'environnement PVC.

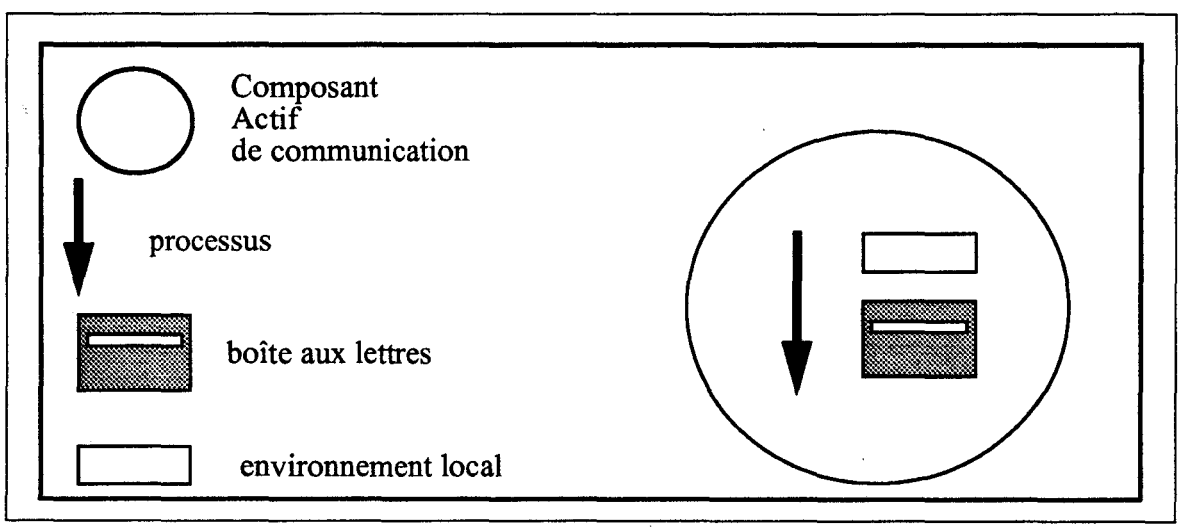


FIG. 2.1 - *Structure d'un Composant Actif de Communication*

Les Cacs introduisent un grain unique de distribution des activités et des données qui va permettre de prendre en charge un parallélisme massif (chaque Cac étant autonome). Cette structure de base va permettre l'implantation de différents modèles d'applications parallèles et notamment des modèles Acteurs ou à Objets Actifs.

2.2.1.2 Les caractéristiques d'un Cac

Dans le cadre de l'étude de la répartition de la charge d'une application utilisant les entités Cacs, il est important de préciser d'une part les caractéristiques d'un Cac qui se déduisent de sa définition et d'autre part des limitations qu'elles imposent.

- Un Cac exécute un comportement

  Le programmeur découpe son application en un ensemble de comportements autonomes. Chaque comportement est une fonction C. La complexité du comportement en terme de calcul ou de communication peut donc être très variable et dépend exclusivement des indications données par le programmeur.

- Un Cac est désigné par un nom unique dans le système qui lui est affecté à la création. Ce nom unique correspond à l'adresse de sa boîte aux lettres.

- Les Cacs disposent d'un espace d'adressage privé, ce qui a pour conséquence d'interdire tout partage de données entre plusieurs Cacs.

- Les Cacs coopèrent par échanges de messages basés sur une communication asynchrone. Un envoi de message se fait en désignant l'adresse de la boîte aux lettres du Cac destinataire. Cette adresse peut être véhiculée comme toute autre donnée, et donc le graphe de communication entre Cacs évolue dynamiquement au cours de l'exécution.

- Un Cac a la possibilité de créer d'autres Cacs en désignant le (ou les) comportement(s) qui sera (seront) exécuté(s). A charge du système ou du programmeur de déterminer le noeud sur lequel sera exécuté le nouveau Cac créé. Le noeud choisi doit disposer localement du code du comportement exécuté par le Cac.

- Un Cac est entièrement localisé sur un noeud de la machine et ne migre pas. La migration est un mécanisme délicat [BSS91, JV89] qu'il n'est pas toujours possible de mettre en place. Les systèmes d'exploitation ne le supportent pas directement en général, il faut alors développer un mécanisme qu'il devient coûteux d'utiliser devant la taille des entités manipulées.

2.2.1.3 Les différents types de comportement

Les comportements manipulés par l'environnement PVC sont issus du découpage d'une appplication écrite dans un langage parallèle à objets. De ce fait les comportements héritent de particularités qui sont fonction de la méthode utilisée pour représenter les objets du langage et de la fonction de l'objet représenté

dans l'application. Nous avons clairement identifié un certain nombre de type de comportements pour lesquels nous avons relevé des caractéristiques en terme de charge.

- Le représentant d'objet

  Ce comportement représente le point d'accès d'un objet de l'application, son rôle est de recevoir les requêtes à destination de l'objet, comme par exemple l'appel de méthode. A la réception d'une requête le représentant d'objet peut réagir de deux façons :

  - soit il exécute lui même la méthode sur l'objet,
  - soit il lance un processus qui aura la charge d'exécuter la méthode.

  Cette dernière possibilité introduit un parallélisme intra-objet, le représentant d'objet pouvant lancer plusieurs processus exécutant le code d'une méthode en concurrence. Le problème de la synchronisation de l'accès aux données partagées par l'objet n'étant pas étudié de ce rapport, nous ne l'aborderons pas, mais il reste un théme de recherche ouvert [GC92]. Les CROs (Composant Représentant d'Objet) sont caractérisés par une communication relativement importante introduite par la réception des requêtes et l'envoi des résultats de l'exécution de méthodes. Leur coût de calcul peut être faible et induire un taux de création de processus important si le CRO lance un processus pour exécuter à sa place le code de la méthode. Les CROs ont en général une durée de vie assez longue.

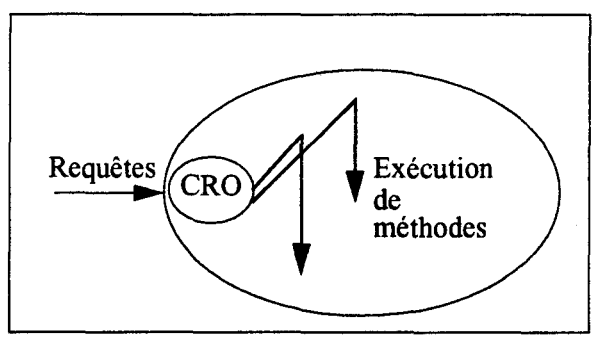


FIG. 2.2 - *Le représentant d'objet*

- Le gérant d'attributs

  Les objets encapsulent à la fois les données et le traitement sur les données. Cependant la représentation de l'objet peut conduire au découpage de cette structure, avec d'un côté le représentant d'objet et de l'autre une entité responsable des attributs de l'objet. Ce découpage peut conduire, dans le cas d'objets stables avec des attributs qui varient peu, à introduire plusieurs

copies des attributs d'un seul objet afin d'assurer un accès simultané aux données. Les CGAs (Composants Gérant d'Attributs) comme les CROs introduisent un coût de communication important. Le coût de calcul étant relativement faible.

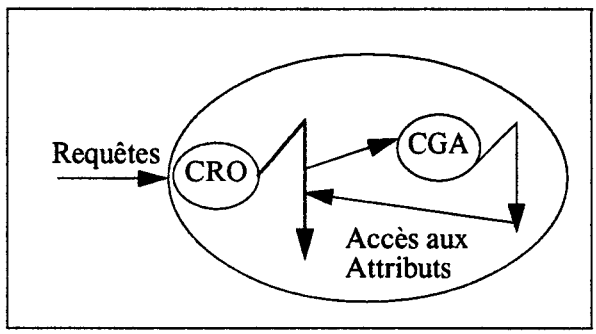


FIG. 2.3 - *Le gérant d'attributs*

- L'exécution de méthode

  Ce type de comportement représente le code d'une méthode ou un groupement de méthodes qui va pouvoir être lancé au cours de l'exécution de l'application. Dans le cas de la réception d'un nombre important de requêtes par un CRO, celui-ci va pouvoir demander la création d'un CEM (Composant d'Exécution de Méthode) afin de créer un parallélisme intra-objet. Les CEM ont comme particularité d'avoir une durée de vie inférieure à la durée de vie des autres processus, ils représentent une charge de calcul relativement importante, et une charge de communication relativement faible.

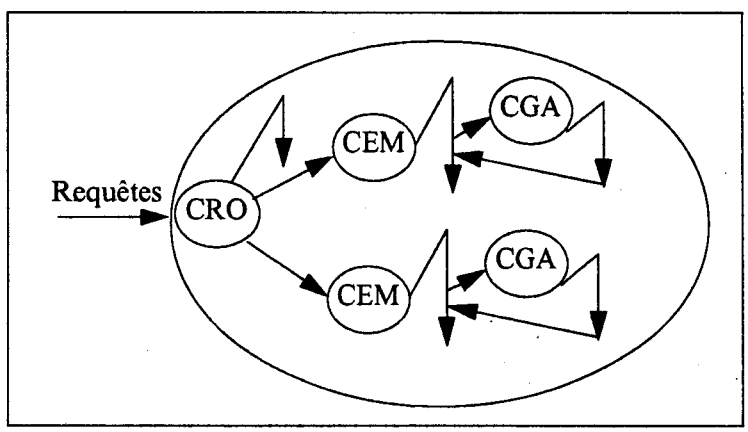


FIG. 2.4 - *L'exécution de méthode*

2.2.2 Présentation du module

Comme il a été indiqué précédemment, un Cac est vu par le système d'exploitation comme un processus léger (thread). Aucun système d'exploitation ne permet la gestion directe de ce type d'entité. Il nous fallait une enveloppe qui permette de les regrouper pour pouvoir les gérer sans pour cela introduire de limitations aux spécifications d'un Cac. L'introduction du **module** correspond à cette notion d'enveloppe vis à vis du système d'exploitation, tout en permettant de définir un premier niveau de répartition de code et par là-même, une répartition de la charge.

2.2.2.1 La structure d'un module

Avant l'exécution, un module est vu comme l'entité de regroupement d'un ensemble de comportements. A l'exécution un module est chargé sur un noeud, il représente un processus qui est géré par le système d'exploitation. Un module va accueillir des Cacs qui vont exécuter les comportements qu'il contient. La création d'un Cac est une requête envoyée à l'un des modules contenant le comportement voulu. Le module (voir Fig. 2.5) se définit comme des données qui représentent essentiellement le code des comportements, une collection de Cacs gérés par un gestionnaire, et un ensemble de serveurs (introduits plus loin) assurant la cohésion de l'ensemble des modules qui constituent l'application.

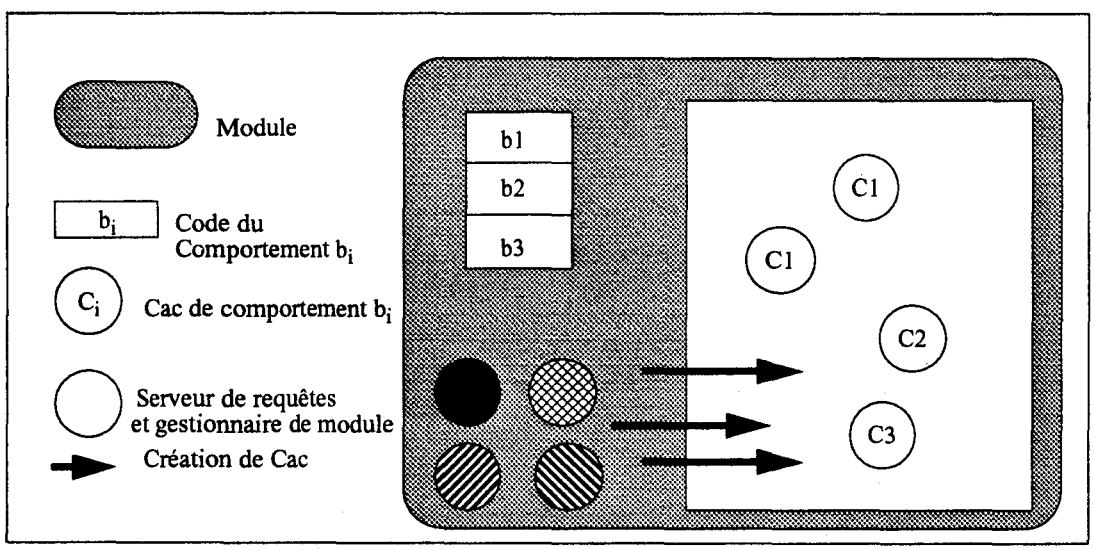


FIG. 2.5 - *Structure d'un Module*

Dans la suite de l'exposé nous considérons qu'il y a au plus un module par noeud. Cette condition ne limite pas les possibilités de la répartition des comportements, il suffit d'ajouter une opération de fusion entre les modules qui auraient

été chargés sur le même noeud, cela de façon à aboutir a un nombre de modules inférieur ou égal au nombre de noeuds de la machine.

2.2.2.2 Les services assurés par les modules pendant l'exécution

Dans l'exécution d'une application aucun module n'a de rôle particulier par rapport aux autres. Ils assurent les mêmes fonctionnalités, seul l'ensemble du code des comportements qu'ils contiennent peut être différent, l'un d'entre eux ayant la particularité de contenir le comportement qui lance l'application.

Le gérant de module

Au chargement de l'application chaque module communique son adresse aux autres modules. C'est cette adresse qui sera utilisée pour faire des requêtes au module. Le gérant de module prend notamment en charge les demandes de création de comportement. Pour cela le gérant de module connaît l'ensemble des comportements qu'il est capable d'exécuter, de même qu'il connaît l'ensemble des modules capables de prendre en charge un comportement particulier. Cette information ne sera pas modifiée au cours de l'exécution de l'application.

Le déverminage

Dans chaque module est ajoutée une activité permettant d'assurer une réexécution déterministe. Dans une phase d'enregistrement de l'exécution ce Cac particulier a la charge de stocker les événements significatifs et ensuite de les envoyer dans la mémoire de stockage. Dans une deuxième phase il permet de contrôler la réexécution et de détecter des conditions distribuées associées à des points d'arrêts [RCM93].

Le ramasse miettes

Les Cacs étant créés dynamiquement, la récupération de ces entités actives quand elles n'ont plus raison d'être, nécessite la mise en place d'un ramasse miettes. L'algorithme de Kafura [KW90] a été implanté en l'adaptant à un environnement réparti en faisant coopérer des Cacs spécialisés présents dans chacun des modules [Dum92].

La répartition de charge

Comme indiqué dans le chapitre 1, une des composantes d'un algorithme de répartition dynamique est l'échange d'informations de charge entre les noeuds de la machine de façon à pouvoir choisir le meilleur lors d'une demande de création. Un Cac spécialisé placé dans chaque module a la charge de fournir ce service d'échange.

2.2.3 Exécution d'une application

L'exécution de l'application se fait en deux phases :

- Le chargement des modules sur les noeuds de la machine et démarrage des serveurs de requêtes inter-modules permettant notamment la création de processus à distance.

- Le lancement proprement dit de l'application en lancant l'exécution d'un comportement particulier *Root* (le comportement *Root* est équivalent de la fonction *main* dans un programme C classique)

Chaque module est chargé sur un noeud de la machine. Le choix du noeud où est chargé un module peut être laissé au programmeur. Dans le cas où la machine est hétérogène, ou laissé à la charge du système, dans ce cas les modules sont placés cycliquement sur les noeuds de la machine. Une fois les modules chargés et les serveurs de requêtes en attente, l'application est exécutée par le comportement *Root* qui va lui-même lancer d'autres comportements. Le lancement d'une application se caractérise par une forte demande de création de processus qui correspond au lancement des objets principaux de l'application.

2.2.4 Environnements matériels supportés

Le run time Cac a été porté sur deux plate-formes matériel [CRGM92] :

- Sur un multicomputer : le MultiCluster II de chez *Parsytec* [Gmb90], cette machine se compose de 32 noeuds, qui sont constitués d'un *Transputer T800* associé à une mémoire locale de 2 MOctets. Chaque noeud peut communiquer avec 4 voisins en utilisant les liens bi-directionnels du Transputer. Le réseau d'interconnexion est complétement reconfigurable. Le run-time utilise le système d'exploitation Helios disponible sur la machine, ce système est un UNIX-like.

- Sur un réseau de stations Sun : chaque station de travail constitue un noeud de l'architecture parallèle distribuée. Le run-time a été développé au dessus du système d'exploitation SunOS (Unix BSD). La communication inter-noeuds utilise le mécanisme des *Sockets* à travers un réseau Ethernet.

- Sur un réseau de Dec-alphas : Nous envisageons l'implantation du run-time PVC sur une *ferme-ALPHA* s'articulant autour d'un *giga-switch.*

2.2.5 Conclusion sur l'environnement PVC

Afin de permettre la conception et l'inplantation de langages parallèles orientés objets, le projet PVC propose le composant actif de communication, entité active d'un grain fin équivalente d'un processus léger (*thread*). Cette structure unique englobant traitement et données s'exécute de manière autonome et concurrente, elle permet de décrire le comportement parallèle d'une application s'exécutant sur une machine de type multicomputer.

Les Cacs sont regroupés en module, qui permet de partager le code de l'application. Ces modules sont répartis sur l'ensemble des noeuds de la machine, ils constituent les entités visibles par le système d'exploitation et intégre un certain nombre de services permettant une gestion inter-modules de l'application et notamment la création à distance de Cacs, le ramasse-miettes, le déverminage, et l'échange d'information de charge.

Dans la suite de cet exposé nous allons détailler notre participation au projet PVC pour intégrer un mécanisme de placement des Cacs.

2.3 Mise en place de la répartition de charge d'une application dans l'environnement PVC

La répartition d'une application va se faire en deux étapes :

1° La répartition statique du code des comportements dans les modules,

2° La répartition dynamique des Cacs au cours de l'exécution de l'application.

Nous allons maintenant présenter comment nous abordons ces deux étapes.

2.3.1 Le placement du code des comportements dans les modules

Ce placement s'inscrit dans une étape du développement de l'application. Il contribue à élaborer le code de chaque exécutable qui sera chargé sur un noeud de la machine cible. Notre objectif est de permettre une exécution la plus courte possible de l'application tout en respectant les contraintes fournies par l'utilisateur ou relatives à l'architecture cible.

La solution idéale est de pouvoir dupliquer l'ensemble du code des comportements dans l'ensemble des modules, ce qui évite les pertes dues au défaut de code à l'exécution. Cette solution devient irréalisable quand le nombre de comportements devient grand. Dans le cas d'une architecture hétérogène certains comportements qui demandent des ressources spécifiques induisent de fait une limitation de la duplication dans les modules qui seront chargés sur les noeuds qui disposent des ressources demandées. Enfin des contraintes spécifiées par l'utilisateur peuvent interdire l'exécution d'un Cac sur un noeud particulier.

Il faut donc trouver un moyen de regrouper les comportements en modules de façon à tenir compte des recommandations de l'utilisateur et de l'architecture cible. Les objectifs attendus par ce regroupement sont les suivants :

- Limiter les défauts de code d'un comportement qui entre en conflit avec l'algorithme de placement dynamique.
- Intégrer toutes les contraintes physiques spécifiées par l'utilisateur.
- Limiter l'encombrement de la mémoire d'un noeud par le code des comportements inutilisés.
- Favoriser une limitation du coût de communication par un regroupement de comportements sur des noeuds voisins.

Après avoir déterminé le nombre de modules N qui vont être créés (N correspond en général au nombre de noeuds de l'architecture), leur localisation physique sur la machine, et l'espace destiné à accueillir le code des comportements pour chaque module T_k, nous allons choisir leur contenu. Notre technique de regroupement va donc se dérouler de la manière suivante:

1° On duplique le code des comportements dans l'ensemble des modules,

2° Si un ou plusieurs comportements ont des besoins particuliers incompatibles avec la situation physique du module sur la machine, alors on le (les) retire de la liste des comportements du module.

3° On applique les recommandations données par le programmeur

A la suite de quoi deux situations peuvent se présenter :

1° Les contraintes d'espace sont respectées c'est à dire :

$\sum_i q_{i_k} \leq T_k$, pour tout k, avec q_{i_k} encombrement du comportement i dans le module M_k.

Et dans ce cas le regroupement en modules obtenu est celui qui sera utilisé pour exécuter l'application.

2° Les contraintes d'espace ne sont pas respectées pour certains modules, et il va falloir chercher une configuration pour ceux-ci qui assure que :

- l'exécution de l'application soit possible (au moins un exemplaire de chaque comportement);
- la distribution obtenue soit la moins pénalisante possible pour l'exécution (maximiser le nombre de duplication du comportement);
- la distribution des comportements minimise les coûts de communication induits par le défaut de localité.

La méthode utilisée pour trouver la composition idéale du code des modules ne satisfaisant pas les contraintes d'espace consiste à formaliser le problème de façon à obtenir une fonction de coût et ensuite de déterminer les solutions qui minimisent cette fonction de coût.

2.3.1.1 Formulation d'une fonction de coût

Cette fonction de coût doit fournir une solution qui remplisse les conditions proposées plus haut. Les informations requises pour son calcul sont :

- Le coût de stockage du code d'un comportement
- Temps d'exécution d'un comportement
- Relation de communication entre les comportements
- Le nombre d'exemplaires d'exécution d'un comportement

Pour obtenir ces informations il y a différentes possibilités (modélisation, simulation, réexécution). Nous avons développé un simulateur (Voir Chapitre 3) qui à l'aide d'un langage de GENEration d'une Suite d'Evénements (GENESE) décrivant l'exécution d'une application, permet d'obtenir une approximation des informations sans avoir à disposer de la machine cible, et donc sans devoir exécuter préalablement l'application.

Nous supposerons donc disposer de la liste des informations suivantes grâce entre autre au simulateur :

- n_{i_k} : le nombre d'exemplaires d'un comportement i exécutés sur le noeud k, avec $N_i = \sum_k n_{i_k}$
- e_{i_k} : Le coût d'exécution d'un comportement i sur le noeud k
- q_{i_k} : Le coût de stockage d'un comportement i sur le noeud k (obtenu à la compilation)
- nc_{ij} : Le nombre de messages échangés entre les N_i comportements i et les N_j comportements j
- C_{kl} : Le coût de communication unitaire entre le noeud k et le noeud l

Remarque sur les hypothèses de travail : Dans cette phase de regroupement du code des comportements en modules, on considère que l'exécution de l'ensemble des Cacs est possible dès le lancement de l'application, et qu'il n'y a pas de contraintes de dépendances. Cela représente le cas le plus défavorable en terme d'occupation mémoire, et même en terme de coût d'exécution et de coût de communication, car il est évident que les exemplaires d'un comportement ne sont pas tous présents en même temps. Cette hypothèse de travail ne rentre pas en conflit avec le but du regroupement de code en modules, qui est de fournir un placement qui ne soit pas pénalisant vis à vis de l'algorithme de placement dynamique qui sera utilisé par la suite.

Le coût d'exécution

Si on considère que l'ensemble des exemplaires d'un Cac (N_i) s'exécutent sur le même noeud (k) alors le coût d'exécution est $N_i e_{i_k}$. Considérons maintenant que l'ensemble des exécutions soient réparties sur deux noeuds k et l. On suppose que les coûts d'exécution de ces Cacs sont indépendants: le coût d'exécution devient alors $\max_k(n_{i_k} e_{i_k})$.

Le coût de communication

Si deux Cacs i et j communiquent nc_{ij} messages unitaires, le coût de communication entre les deux Cacs va dépendre de leur placement respectif, et du coût unitaire de communication entre les noeuds qui les contiennent :

$$\sum_k \sum_l \frac{n_{i_k} n_{j_l} C_{kl} nc_{ij}}{N_i N_j}$$

La fonction de coût à minimiser

La fonction de coût à minimiser pour obtenir un placement des comportements dans les modules qui soit le moins pénalisant est la suivante :

$$\underbrace{\alpha \sum_{i} \max_{k}(n_{i_k} e_{i_k})}_{(1)} + \underbrace{\beta \sum_{\substack{i,j \\ j \geq i}} \sum_{k} \sum_{l} \frac{n_{i_k} n_{j_l} C_{kl} nc_{ij}}{N_i N_j}}_{(2)}$$

avec les contraintes suivantes :

$$\sum_{k} n_{i_k} = N_i$$

$$\sum_{i} q_{i_k} \leq T_k \text{pour tout module } M_k$$

La partie (1) de la fonction tend à répartir le plus possible les exemplaires de l'ensemble des comportements. En effet, plus il y aura de noeuds possibles et plus la valeur de $\max_{k}(n_{i_k} e_{i_k})$ sera petite. La partie (2) cherche à regrouper les comportements exécutés sur des noeuds voisins. Les coefficients α et β, quant à eux, permettent de relativiser les deux parties de la fonction de coût.

2.3.1.2 Description de l'algorithme de recherche des solutions minimisant la fonction de coût

Les algorithmes de recherche des solutions minimisant une fonction de coût sont légion. Ils sont plus ou moins coûteux en temps, comme indiqué dans le chapitre I, il faut faire le choix entre une solution optimale, ou une bonne solution. Nous utilisons des données obtenues par simulation qui sont donc relatives aux paramètres fournis, et aux données d'entrées.

C'est pour ces raisons que nous avons choisi un algorithme qui développe une heuristique et en particulier la méthode du recuit simulé.

La méthode du recuit simulé

La méthode du recuit simulé est dérivée d'un principe de thermodynamique qui veut que par abaissement progressif de la température sur un ensemble de particules en interaction on obtient une configuration à énergie minimale. Quand la température T est élevée, cela provoque une agitation ponctuelle du système.

Ce phénomène est utilisé pour éviter de bloquer la recherche d'une solution sur un minimum local. Il se traduit par l'évolution d'une solution de la manière suivante :

Soit E le coût de la solution retenue à un instant donné, soit E_i le coût de la solution évaluée, et soit $\Delta E = E_i - E$ la différence entre les deux coûts. Si $\Delta E \leq 0$ alors la nouvelle solution est acceptée avec la probabilité 1, sinon la nouvelle solution est acceptée avec une probabilité $\exp^{\frac{-\Delta E}{T}}$.

Plusieurs paramètres sont utilisés

- $Tmax$: La température de départ. $Tmax > 0$
- $Tmin$: La température de fin. $Tmax > Tmin > 0$
- a : La vitesse de décroissance de la température. $0 \leq a < 1$
- Nb_Sol_Acc : Nombre de solutions acceptées. $Nb\_Sol\_Acc \geq 0$
- Nb_Sol_Prop : Nombre de solutions proposées. $Nb\_Sol\_Prop > 0$

L'algorithme

On retrouve dans la figure 2.6 la boucle principale de l'algorithme du recuit simulé qui a été implémenté. La fonction *générer_solution_initiale()* donne une première solution avec une distribution du code des comportements dans les modules qui satisfait les contraintes d'espace pour chaque module, en reprenant le contenu des modules qui répondent aux contraintes d'espace et en modifiant le contenu des modules qui n'y répondent pas.

La fonction *"générer_solution(i,S_{crt})"* est utilisée pour générer une nouvelle solution. Cette fonction est très importante car elle conditionne l'espace des solutions qui vont être examinées et parmi lesquelles on choisira celle qui minimise la fonction de coût. Nous avons choisi d'équilibrer la génération de solutions qui tentent de réduire la duplication des comportements avec celles qui tentent d'augmenter la duplication. Pour cela la fonction *"générer_solution(i,S_{crt})"* va chercher alternativement un noeud en plus pour le comportement i ou un noeud en moins. Un module $M_{aléa}$ est choisi aléatoirement, celui-ci dispose d'un ensemble de codes de comportement Ω. Deux cas sont possibles :

- Soit *"générer_solution(i,S_{crt})"* cherche à réduire le nombre de modules contenant le comportement i. Dans ce cas on retire le code du comportement i du module $M_{aléa}$ ($M_{aléa}$ tel que $code(i) \in \Omega_{aléa}$) chargé sur le noeud $P_{aléa}$, et on redistribue les Cacs qui exécutent le comportement i uniformément sur les autres noeuds ayant chargé un module contenant le code du comportement i.

```
debut
    lire configuration d'une application
    lire les paramètres de l'algorithme de recuit simulé
    S := générer_solution_initiale()
    E := calcul_coût_solution(S)

    tq T ≥ Tmin faire
      pour i = 0 jusqu'a i < MaxCac faire
            j := 1
            k := 1
            tq j ≤ Nb_Sol_Acc ∧ k ≤ Nb_Sol_Prop faire
              générer_solution(i, S_crt)
              E_i = calcul_coût_solution(S_crt)
              ΔE := E_i − E
              si ΔE ≤ 0 alors
                sauve_solution(S_crt, S)
                E := E_i
                k := k + 1
                j := j + 1
              sinon
                    p_augmentation := exp (−ΔE / T)
                    P := random
                    si P ≤ p_augmentation alors
                      sauve_solution(S_crt, S)
                      E := E_i
                      k := k + 1
                      j := j + 1
                    sinon
                          k := k + 1
                    fsi
              fsi
            fait
            i := i + 1
      fait
      T := T * a
    fait
    Donne_solution(S)
fin
```

FIG. 2.6 - *Présentation de l'algorithme du recuit simulé*

- Soit *"générer_solution(i,S_{crt})"* cherche à augmenter le nombre de modules contenant le comportement i. Dans ce cas on ajoute le code du comportement i dans le module $M_{aléa}$ ($M_{aléa}$ tel que $code(i) \notin \Omega_{aléa}$) chargé sur le noeud $P_{aléa}$, et on redistribue uniformément les Cacs qui exécutent le comportement i sur l'ensemble des noeuds ayant chargé un module contenant le code du comportement i.

debut
  si *Ajout* **alors**
    $aléa = ch\_mod\_sans\_code(i)$
    $\Omega_{aléa} = \Omega_{aléa} \cup code(i)$
    $redistribution\_des\_instances(i)$
  sinon
    $aléa = ch\_mod\_avec\_code(i)$
    $\Omega_{aléa} = \Omega_{aléa} - code(i)$
    $redistribution\_des\_instances(i)$
  fsi
fin

La solution rencontrée qui minimise la fonction de coût conditionne la répartition finale du code des comportements dans chacun des modules.

2.3.1.3 Exemple

Simulation d'une station de lavage

Pour illustrer notre implantation du regroupement des comportements par modules, nous allons utiliser l'exemple de la simulation d'une station de lavage.

Cet exemple sera présenté dans la suite de ce rapport. Dans les tableaux qui suivent nous allons donner les informations nécessaires pour évaluer le regroupement des comportements (Tab. 2.1 et 2.2). Ces informations ont été obtenues grâce au simulateur qui est présenté dans le chapitre 3. Si l'on suppose que chaque module ne dispose que d'un volume fini (fixé à 50U dans l'exemple) pour stocker le code des comportements, la duplication dans chaque module de l'ensemble des comportements n'est pas possible. Il faut alors rechercher une solution qui minimise la fonction de coût présentée précédemment. Les résultats obtenus après recherche de la solution minimale de la fonction de coût grâce à la méthode du recuit simulé sont donnés dans le tableau 2.3. Les résultats obtenus suite à une simulation utilisant un regroupement aléatoire et le regroupement obtenu après recherche heuristique sont présentés dans le tableau 2.4 et montrent une amélioration du temps de simulation dans le cas d'une utilisation d'un regroupement

calculé, et cela pour un placement dynamique en aveugle (Cyclique) ou utilisant les informations de charge dynamique du système (Min_IC).

2.3.2 Le placement dynamique des Cacs

Comme nous l'avons vu dans la présentation du système PVC, l'exécution d'une application propage sa charge en créant dynamiquement des processus à partir du code des comportements qui existent en plusieurs exemplaires distribués sur des noeuds différents de la machine. De plus, notre système permet la transmission du nom unique d'un processus par message, ce qui permet de modifier dynamiquement le graphe des relations qui existent entre les processus. Les mécanismes de placement statique qui reposent sur une connaissance préalable à l'exécution de l'ensemble des tâches qui seront lancées, des graphes de dépendances et de comunication, ne peuvent permettre un placement efficace de notre application.

Nous avons développé un mécanisme de répartition dynamique des processus pour le système d'exécution PVC. Chaque comportement est disponible dans un ensemble de modules et donc a la possibilité d'être exécuté sur un ensemble de noeuds. Le travail de la stratégie de répartition dynamique est de trouver le meilleur noeud pour exécuter le comportement demandé. La notion du meilleur noeud pour nous étant celui qui conduira à un temps d'exécution minimal pour l'application. Il convient donc de répartir au mieux la charge de calcul sans augmenter de manière préjudiciable le coût de communication.

2.3.2.1 Présentation du contexte pour une stratégie de placement dynamique

Pour développer une stratégie de placement dynamique, il nous faut définir les caractéristiques des applications et des processus que l'on va devoir répartir.

- Le taux de création dynamique.

  Les applications que nous allons être amenés à répartir provoquent la création d'un nombre important de processus. Le taux de création va être généralement important au lancement de l'application qui correspond à la phase de création des représentants d'objets et des structures de données actives. Ensuite dans une phase d'exécution on rencontre plutôt des demandes de création de processus qui vont exécuter des méthodes sur les objets pour le compte des représentants d'objets et qui utiliseront les données stockées dans les structures de données actives. Nous nous situons donc dans une problématique où l'outil de décision sur la répartition des processus à créer va être très souvent sollicité.

Nom	Nombre	Coût d'exécution	Coût de stockage								
			P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Root	1	7734	50	-1	-1	-1	-1	-1	-1	-1	-1
Tronçon	7	2936	10	10	10	10	10	10	10	10	10
Contrôleur	3	4225	10	10	10	10	10	10	10	10	10
Voiture	20	2174	10	10	10	10	10	10	10	10	10
Poste-Lavage 0	7	139	10	10	10	10	10	10	10	10	10
Poste-Lavage 1	6	261	10	10	10	10	10	10	10	10	10
Poste-Lavage 2	5	157	10	10	10	10	10	10	10	10	10
Poste-Lavage 3	5	84	10	10	10	10	10	10	10	10	10

TAB. 2.1 - *Coûts d'exécution et de stockage pour chaque comportement*

	Nombre de messages échangés entre les Comportements							
	Root	Tronçon	Contrôleur	Voiture	Poste-Lav. 0	Poste-Lav. 1	Poste-Lav. 2	Poste-La. 3
Root	0	7	3	20	0	0	0	0
Tronçon	7	0	219	862	0	0	0	0
Contrôleur	3	223	0	0	7	6	5	5
Voiture	20	862	0	0	6	5	5	4
Poste-Lavage 0	0	0	7	6	0	0	0	0
Poste-Lavage 1	0	0	6	5	0	0	0	0
Poste-Lavage 2	0	0	5	5	0	0	0	0
Poste-Lavage 3	0	0	5	4	0	0	0	0

TAB. 2.2 - *Communications constatées entre comportements*

	Nombre de comportements par noeud								
	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Root	1	0	0	0	0	0	0	0	0
Tronçon	0	0	1	1	1	1	1	1	1
Contrôleur	0	0	0	0	0	0	1	1	1
Voiture	0	0	3	3	3	3	3	3	2
Poste-Lav. 0	0	2	2	2	1	0	0	0	0
Poste-Lav. 1	0	0	0	1	1	1	1	1	1
Poste-Lav. 2	0	0	2	2	0	1	0	0	0
Poste-Lav. 3	0	0	0	0	1	1	1	1	1

TAB. 2.3 - *Solution obtenue par recherche heuristique*

	Cyclique	Min_IC
Av. R.S.	24992	21624
Ap. R.S.	23372	20845

TAB. 2.4 - *Résultats de simulation*

- Granularité des processus

  Nous nous situons dans un contexte où les processus à placer sont d'un grain fin. Comme indiqué dans la présentation de l'environnement PVC, les processus qui sont créés dynamiquement sont des processus légers (thread) au sens UNIX du terme. La phase de placement pour chaque processus doit donc être la plus courte possible pour apporter un gain sur son exécution, et elle se justifie d'autant plus que le nombre de créations des processus est important. La réponse au choix d'un noeud à chaque demande de création doit être la plus rapide possible. Dans ce contexte il vaut mieux avoir une stratégie de placement rapide mais pas toujours juste plutôt qu'une stratégie de placement optimale mais qui demande un temps de réponse qui va perturber l'application.

- Gamme des processus à placer

  Les processus à placer n'ont pas tous les mêmes caractéristiques, comme nous l'avons indiqué (voir 2.2.1.3). Certains processus engendrent plutôt un trafic important en communication, et d'autres provoquent une utilisation de la CPU qui est importante. La stratégie doit donc adapter son placement en fonction d'un type de comportement.

Etant données les caractéristiques des entités que nous avons à placer et le modèle d'exécution utilisé, nous avons choisi de développer des stratégies de répartition dynamique qui utilisent un algorithme de décision local à chaque noeud. A cela, nous avons associé un algorithme d'échanges d'informations (dans le cas où la charge de la machine est utilisée pour répartir les processus) qui duplique l'état global de la machine sur chaque noeud. Le placement préemptif des processus n'a pas été envisagé, la migration d'un processus léger n'existe pas sur les systèmes d'exploitations utilisés, de plus le coût nécessaire à sa mise en place aurait été trop important en général par rapport au temps d'exécution du processus. Ces choix permettent de garantir un temps de réponse le plus court possible (sans communication) pour le choix d'une localisation pour un processus à créer. De plus ils permettent d'assurer une certaine tolérance aux pannes grâce à la duplication des informations sur l'état global de la machine.

L'exécution d'une application varie en fonction des comportements qui la composent. Les besoins en termes de ressources (communication, CPU), le taux de création dynamique de processus, le nombre de comportements et leurs types, l'architecture de la machine et les caractéristiques des noeuds qui la composent sont autant de facteurs qu'il faut prendre en compte pour choisir la stratégie de répartition la plus adaptée pour une application donnée. A cela peut venir s'ajouter des spécifications données par le programmeur, qu'il faut pouvoir intégrer au moment du choix d'une localisation d'un processus.

Certaines applications qui font des créations de processus uniquement dans la phase de lancement, n'ont pas besoin d'un algorithme de répartition dynamique puissant. D'autres qui contiennent un nombre important de comportements avec des types différents, des besoins en ressources très variables vont nécessiter la mise en place d'algorithmes de placement plus élaborés.

Dans la suite, après avoir présenté comment est traitée une demande de création, nous présenterons les différents mécanismes que nous avons développé pour permettre une répartition qui satisfasse aux particularités de l'application et de l'architecture. Ensuite nous présenterons comment le mécanisme a été introduit dans l'environnement PVC et pris en compte par l'application.

2.3.2.2 Traitement d'une demande de création d'un processus

Etant donné que nous n'envisageons pas de déplacer un processus au cours de son exécution, la phase de répartition de charge se situe essentiellement à la création des processus. L'algorithme décrit à chaque demande de création est présenté dans la figure 2.7.

```
debut
        ok ← Faux
        tq ¬ok faire
            n ← recherche_du_meilleur_noeud(Cp)
            envoi_demande_création(n, Cp)
            attente_réponse(n, nom_unique_proc)
            si nom_unique_proc ≠ Erreur alors
                ok ← Vrai
            fsi
        fait
fin
```

FIG. 2.7 - *Traitement d'une demande de création*

Si un processus P_1 qui exécute le comportement C_1 fait une demande de création d'un processus P_2 pour qu'il exécute un comportement C_2 alors le processus P_1 va exécuter la procédure de recherche du meilleur noeud pour exécuter le processus P_2. Cette procédure s'exécute localement, elle utilise suivant la stratégie de répartition utilisée des informations comme :

- Les informations de charge de la machine,
- Le comportement à exécuter,

- Le type de comportement à exécuter (précise les ressources qui vont être consommées par le comportement : CPU, mémoire, communication).

Ou bien encore la stratégie de répartition n'utilise aucune information, dans ce cas le choix est fait de manière aléatoire ou cyclique.

Après avoir déterminé le numéro du noeud n qui va accueillir le processus P_2, on envoie un message au gérant de module, responsable des créations, du module qui a été chargé sur le noeud n. Le processus P_1 se bloque en attente de la réception du nom unique du processus créé. Dans notre implantation le noeud qui a été choisi ne peut pas refuser la création sauf dans le cas d'un dépassement de capacité mémoire. Dans ce cas le gérant de module renvoie un message d'erreur et la procédure de recherche du meilleur noeud est relancée en s'interdisant le noeud n.

2.3.2.3 Intégration d'un mécanisme de répartition dynamique dans PVC

Les applications que nous avons à placer et les comportements qui les composent ont des exécutions très variables en coût de communication, ou en coût de calcul. Le choix d'une stratégie de placement qui privilégierait un type de comportement d'exécution plutôt qu'un autre, aurait pour conséquence de réduire l'ensemble des applications qui peuvent être traitées. Le choix d'une architecture de machine et d'une topologie particulière aurait pour conséquence de réduire là aussi l'utilisation de notre système d'exécution. Pour toutes ces raisons nous avons choisi de développer un ensemble de stratégies de placement paramétrables en fonction de l'application à placer, mais aussi de l'architecture de la machine utilisée.

La technique consiste à fournir aux programmeurs le choix d'une stratégie de placement. Cette stratégie, l'utilisateur la construit en définissant les quatres composantes qui ont été présentées dans le paragraphe 1.6.2.5. Le programmeur indique :

- La manière de calculer la charge d'un noeud,
- La politique de transfert,
- La politique d'information,
- La politique de localisation.

Dans un fichier de paramètres il a la possibilité d'affiner le comportement de la stratégie de placement qu'il bâtit en fonction de l'application et de la machine utilisée.

Nous allons maintenant présenter ce qui a été implanté dans le run time PVC en reprenant une par une les quatre composantes.

2.3.2.3.1 La politique de calcul de la charge d'un noeud. Nous avons implanté trois politiques de calcul différentes pour indiquer la charge d'un noeud.

- **Le nombre de processus présents.** Chaque processus quelque soit son état est comptabilisé dans le calcul de la charge d'un noeud. Le programmeur a la possibilité de spécifier un coût relatif pour chaque comportement. Si l'utilisateur ne dispose pas d'informations sur les comportements il indique une valeur unitaire pour chacun des comportements. Dans ce cas l'algorithme de répartition ne fait pas de distinction entre tel ou tel comportement. Sinon le coût de chaque comportement est intégré dans le calcul de charge.

- **Le nombre de processus prêts.** Cette fois seuls les processus qui sont en attente de la CPU sont comptabilisés dans la charge du noeud. De la même manière que pour l'indicateur de charge précédent le programmeur à la possibilité de spécifier un coût relatif pour chaque comportement.

- **Le nombre de messages en attente.**

Comme nous l'avons indiqué dans le chapitre 1, le choix d'un indicateur de charge représentatif n'est pas trivial. Son choix repose sur l'activité de l'application à placer. Si l'application se compose de processus plus généralement en attente de message, on trouvera une corrélation significative entre la charge future et les processus prêts. Pour les applications qui communiquent beaucoup le nombre de messages en attente de traitement peut être un bon indicateur de la charge future sur le noeud.

2.3.2.3.2 La politique de transfert. Dans notre implantation l'algorithme de placement n'intervient qu'au moment de sa création. La migration d'un processus n'étant pas envisagé, la politique se résume à choisir entre le partage de charge et l'équilibrage de charge. Dans le cas du partage de charge, le programmeur peut spécifier une valeur de seuil limite au-dessus de laquelle la création sur un noeud distant est envisagée. On peut choisir de faire évoluer dynamiquement la valeur du seuil, dans ce cas la valeur du seuil sur un noeud est recalculée en fonction des informations de charges disponibles sur les autres noeuds. La valeur du seuil est alors la moyenne de la charge sur l'ensemble des noeuds. Cette configuration se rapproche plus d'un équilibrage de charge, mais où la création distante est moins systématique.

2.3.2.3.3 La politique d'information. Les gains apportés par une régulation dynamique de la charge sont assez sensibles au choix de la politique d'information. En effet sa mise en place n'est pas négligeable, et généralement une bonne politique d'information n'est pas celle qui va fournir l'information de charge la plus pertinente. En effet, le coût d'une décision de localisation basée sur une information optimale est trop important. Nous avons implanté un certain nombre de politiques d'information que nous allons présenter.

- **Broadcast.** Le service dédié à la répartition dynamique échantillonne la valeur de la charge sur le noeud avec une certaine **fréquence** paramétrable. Cette valeur est comparée avec la précédente. Si la différence entre les deux dépasse un certain **seuil** paramétrable, alors il y a émission d'un message vers l'ensemble des autres noeuds pour leur communiquer la nouvelle valeur de la charge. Dans notre implantation le seuil est fixe tout au long de l'exécution, de même que la fréquence.

- **Jeton.** L'information de charge de chaque noeud est rangée dans un vecteur. Celui-ci circule de noeud en noeud en suivant un **chemin** adapté à l'architecture de la machine. Seul le noeud qui contient le jeton peut modifier sa valeur de charge dans le vecteur. Le jeton circule avec une certaine **fréquence**, qui varie en fonction de la variation globale des informations de charges. Si la somme des différences de charges dépasse un certain **seuil** le noeud modifie la fréquence de circulation.

- **Message.** L'information de charge est ajoutée aux messages qui sont envoyés par le processus de l'application, cela permet de ne pas ajouter de coût de communication pour la politique d'information. Un traitement est cependant nécessaire à l'arrivée de chaque message pour prendre en compte la valeur de l'indicateur de charge du noeud émetteur du message.

- **Message Plus.** Cette politique essaye de corriger un inconvénient de la politique d'information précédente. Les échanges d'informations suivent les échanges de l'application, les indicateurs de charges ne dispose donc pas toujours de la valeur la plus récente. En effet, les processus d'un noeud ne communiquent pas forcement avec des processus situés sur l'ensemble des autres noeuds de l'architecture. La valeur de l'indicateur de charge des noeuds qui n'échangent pas de messages (ou depuis un certain temps) pour l'application est donc fausse. Pour corriger cela à chaque demande de création, on choisit cycliquement un ensemble de N noeuds. Pour chacun des noeuds, si la date de modification de l'indicateur de charge est trop ancienne on fait une demande d'information ponctuelle. De cette façon, les informations de charges sont maintenues même si l'application échange peu de messages.

- **Threshold.** A chaque demande de création d'un processus, le site fait des demandes d'informations de charge pour N noeuds choisis de manière cyclique. L'algorithme de recherche du meilleur noeud bénéficiera de cette information lors d'une prochaine demande de création, c'est en quelques sorte une demande d'information en prévision des demandes de création futures.

- **Gradient et Stratégie passive.** Dans le cas du gradient la technique est basée sur une information restreinte de l'état global de la machine (voir chapitre 1 1.6.2.2) qui permet de retrouver de proche en proche le noeud le moins chargé et le plus proche. Comme nous ne faisons pas de migration, l'implantation utilise l'envoi de message pour la remplacer. La détermination du meilleur noeud peut conduire à l'envoi de plusieurs messages (voir chapitre 3 3.2.6.6) ce qui a pour conséquence d'augmenter le temps de réponse de la procédure de recherche du meilleur noeud. Dans le cas de la stratégie passive, un noeud qui atteint un niveau de charge inférieur à un certain **seuil** est considéré comme inactif, et fait une demande de charge vers les noeuds les plus chargés. Comme la migration n'est pas utilisée, là encore le noeud le plus chargé ne peut pas se décharger immédiatement en migrant certains processus, il doit attendre qu'il y ait des demandes de création. Ce décalage peut entrainer une instabilité du système de répartition, tout les sites choisissant comme meilleur noeud le dernier qui a envoyé une demande de travail.

2.3.2.3.4 La politique de localisation. Les différentes politiques de localisation implantées varient par le nombre d'informations qui vont être utilisées pour effectuer le choix d'un noeud pour exécuter un comportement.

- **Aléatoire et cyclique.** L'algorithme de choix du meilleur noeud ne prend en compte aucune information de charge pour déterminer le meilleur site. Un tirage aléatoire est utilisé dans le premier cas, une variable sur chaque noeud qui varie cycliquement de 1 à N (N étant le nombre de noeuds dans la machine), détermine le numéro du prochain *meilleur* noeud.

- **Introduction de l'état global de la machine avec ou sans les informations de coût par comportement.** L'algorithme de choix du meilleur noeud tient compte des informations sur la charge de la machine. Dans ce cas le noeud qui a l'indicateur de charge le moins important est choisi. Si plusieurs noeuds ont la même valeur minimale alors le noeud le plus proche (dans le cas où la notion de distance a un sens) est choisi.

- **Introduction d'une localisation spécifique en fonction du type du comportement.** Le type du comportement va permettre de particulariser

le placement. Les CROs et les CGAs sont placés cycliquement, les CEMs sont placés en utilisant les informations de charge disponibles. Les caractéristiques d'un CRO et d'un CGA font que ces composants sont généralement créés au lancement de l'application. Or dans cette période, l'état global de la machine est difficilement maintenu par un mécanisme d'échanges d'informations: donc pour assurer une répartition équilibrée de la communication on utilise un choix cyclique. Pour les CEMs qui représentent le plus gros du travail de répartition à gérer (mais sur toute la période d'exécution de l'application) on utilise les informations de charge disponibles.

2.3.2.4 Analyse du placement d'une application

Dans ce paragraphe nous allons présenter les variations du temps d'exécution en fonction d'un échantillon des paramètres de répartition dynamique. Pour cela nous allons utiliser l'application factorielle.

Présentation de factorielle. Cette application consiste à calculer la valeur de factorielle n en parallèle (voir Fig. 2.8). L'algorithme divise récursivement en deux sous-calculs $([n, \frac{n+m}{2}], [\frac{n+m}{2} + 1, m])$, le calcul initial $([n, m])$. Le découpage s'arrête lorsque les deux bornes sont égales ($[a, b]$ avec $a = b$), alors la valeur est retournée au calcul de niveau supérieur, celui-ci multiplie les deux valeurs reçues et retourne à son tour le résultat au niveau supérieur, etc .... Le code de l'application est donné en annexe (voir C.1). Ce programme construit un arbre binaire équilibré de processus. Chaque processus créé exécute le même code. Des processus fils communiquent avec le processus père pour lui fournir le résultat. Le nombre d'entités créées est fonction de la valeur de n. Plus la dimension du domaine va être petite (c'est à dire l'intervalle entre la borne inférieure et la borne supérieure du calcul), plus le nombre de demandes de création simultanées va être important. Seules les feuilles de l'arbre sont actives, les noeuds intérieurs attendent le retour des solutions. Pour cette application il n'y a pas de données partagées ou échangées, elles sont toutes fournies à la création de l'entité.

Présentation des conditions de placement Les exécutions ont été faites sur la machine parallèle disponible au LIFL, le MultiCluster II de chez Parsytec (voir le paragraphe 2.2.4).

Les topologies décrites par le réseau de connexion reliant les transputers sont basées sur le modèle du tore avec 4, 9, 16, 20, et 25 Transputers (voir Fig. 2.9).

Présentation des paramètres de répartition utilisés. Pour cette application nous avons choisi de comparer trois politiques d'informations **Broadcast**,

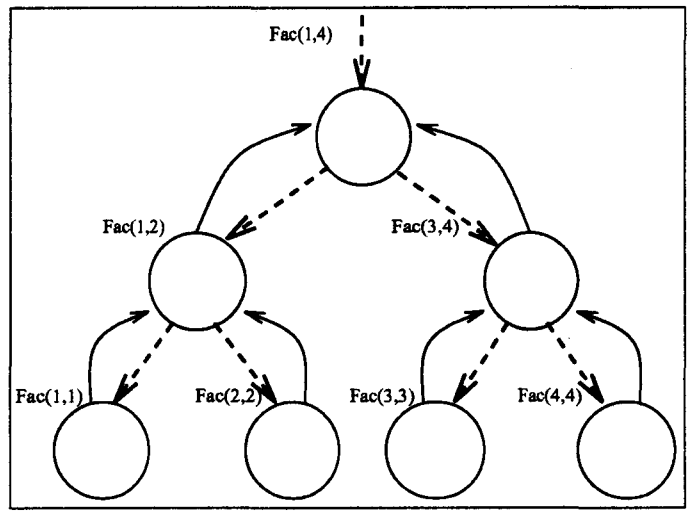


FIG. 2.8 - *Calcul de factorielle par dichotomie*

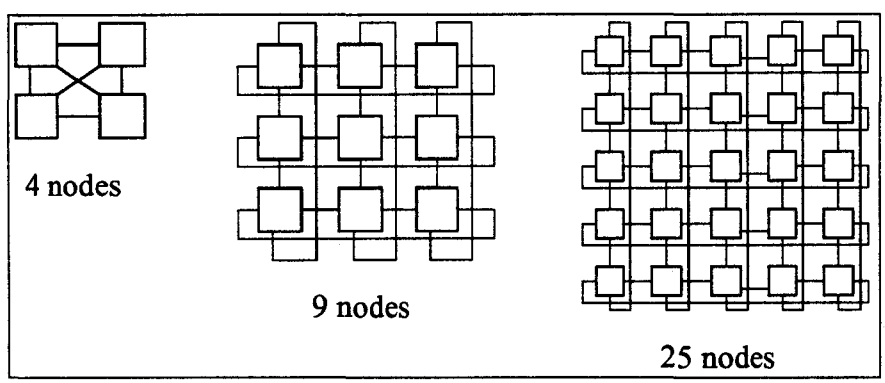


FIG. 2.9 - *Les topologies de réseau utilisées*

Jeton, et **Message** qui utiliseront comme politique de localisation, celle qui consiste à choisir comme meilleur site j, celui pour lequel la distance $D_{i,j}$ (i étant le demandeur de la création) est minimale parmi l'ensemble des sites qui ont une valeur minimale d'indicateur de charge. La valeur de l'indicateur de charge est calculée en utilisant le nombre de Cacs présents sur un site, pondéré par leur coût relatif. L'information concernant le coût des comportements a été obtenue soit par simulation, soit par une précédente exécution. A cela nous avons ajouté deux stratégies de placement en aveugle **Aléatoire** et **Cyclique**.

Modification de l'application factorielle Dans l'application factorielle n, il n'y a qu'un seul comportement. La possibilité d'introduire le coût relatif d'un comportement n'a donc pas d'intérêt, sauf si le coût est directement imputable à chaque instance du comportement qui est créée.

Pour factorielle n par exemple, le coût que représente une instance créée dans

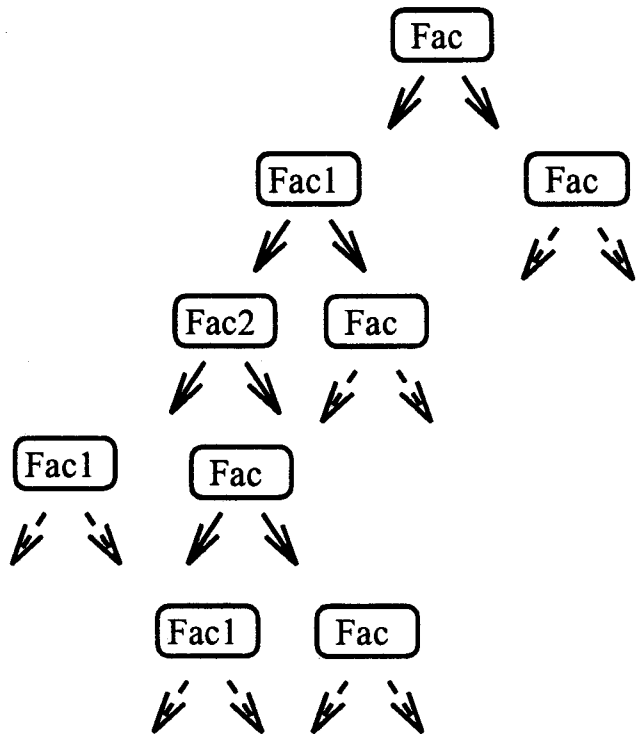


FIG. 2.10 - *Arbre de résolution par factorielle n modifiée*

le début de l'exécution n'est pas le même que dans la fin de l'exécution. En effet si on fait le rapport

$$\frac{\textit{temps d'occupation de la CPU}}{\textit{durée de vie du processus}}$$

Cette valeur va devenir de plus en plus proche de un au fur et à mesure que l'on va descendre dans l'arbre de résolution. On en déduit donc une variation du coût de l'instance d'un même comportement en fonction des données qu'il a à traiter. De là, on peut calculer une fonction de coût en fonction d'un paramètre qui nous indique le coût que va représenter l'instance du comportement créée. Ce calcul est fortement lié aux données de l'application et à l'environnement du comportement dans l'application, il est donc difficilement réutilisable d'une exécution à l'autre.

Nous avons modifié artificiellement l'application factorielle pour introduire des comportements différents. De cette façon nous allons pouvoir étudier le déroulement de l'exécution d'une application lors de la prise en compte d'un coût différencié en fonction des comportements. Pour cela nous avons conservé le déroulement de l'application, et ajouté deux nouveaux pseudo-comportements *fac1 et fac2* (voir Fig. 2.10) auxquels nous allons affecter des coûts de calcul différents.

Analyse des résultats d'exécution de l'application factorielle n. La figure 2.11 indique les temps d'exécution obtenus par les stratégies de placement utilisées en fonction de l'architecture de la machine. Dans la figure 2.11(a), les comportements sont tous identiques, c'est à dire que les coûts de calcul sont les mêmes. Dans la figure 2.11(b), les trois comportements ont des coûts différents ($fac2 = 2fac1 = 6fac$).

Dans la figure 2.11(a), la stratégie de placement en aveugle *Cyclique* donne le temps d'exécution le plus court et cela quelque soit l'architecture de la machine. Les stratégies *Message* et *Jeton* donnent aussi de bons résultats, alors que les stratégies *Aléatoire* et *Broadcast* obtiennent les moins bons résultats. La stratégie de placement *Cyclique* est la plus adaptée pour réguler cette application qui génère une charge parfaitement uniforme, ce qui explique ses résultats. Les stratégies qui engendrent un sur-coût minimal (*Message, Jeton*) pour mettre en place la politique d'information qui permet de réguler en fonction de l'état de charge de la machine donnent des temps d'exécution intermédiaires. La stratégie *Aléatoire* donne des résultats variables mais toujours moins bons que pour les précédentes stratégies, pendant que la stratégie *Broadcast* n'arrive pas à réduire le temps d'exécution globale de l'application lorsque le nombre de noeuds augmente pour la simple raison que le sur-coût de communication augmente lui aussi.

Dans la figure 2.11(b), Les stratégies *Message* et *Jeton* donnent cette fois les meilleurs résultats devant la stratégie *Cyclique*. Le faible coût de la politique d'information permet un placement des processus en fonction de l'état de la machine et du comportement plus judicieux qu'un placement en aveugle. Même si les gains apportés ne paraissent pas important, on montrera (cf chapitre 4 4.4.1) que plus la différence de coût entre les comportements est importante et plus les stratégies en aveugle deviennent mauvaises.

Cet exemple simple de placement nous montre combien il est important de faire évoluer la stratégie de placement dynamique en fonction de l'architecture de la machine parallèle, mais aussi en fonction de l'application.

2.4 Conclusion

Dans ce chapitre nous avons fait une présentation du projet PVC et du système d'exécution qu'il utilise. L'application est vue comme un ensemble de Composants Actifs de Communication qui s'exécutent de manière autonome et concurrente. Les Cacs sont regroupés en modules qui constituent les entités visibles pour le système d'exploitation et mettent en place entre autre des mécanismes de communication et de demandes de création inter-modules.

Le problème de la répartition de l'application sur les noeuds de l'architecture se fait en deux étapes. Dans une première étape qui consiste à placer le code des

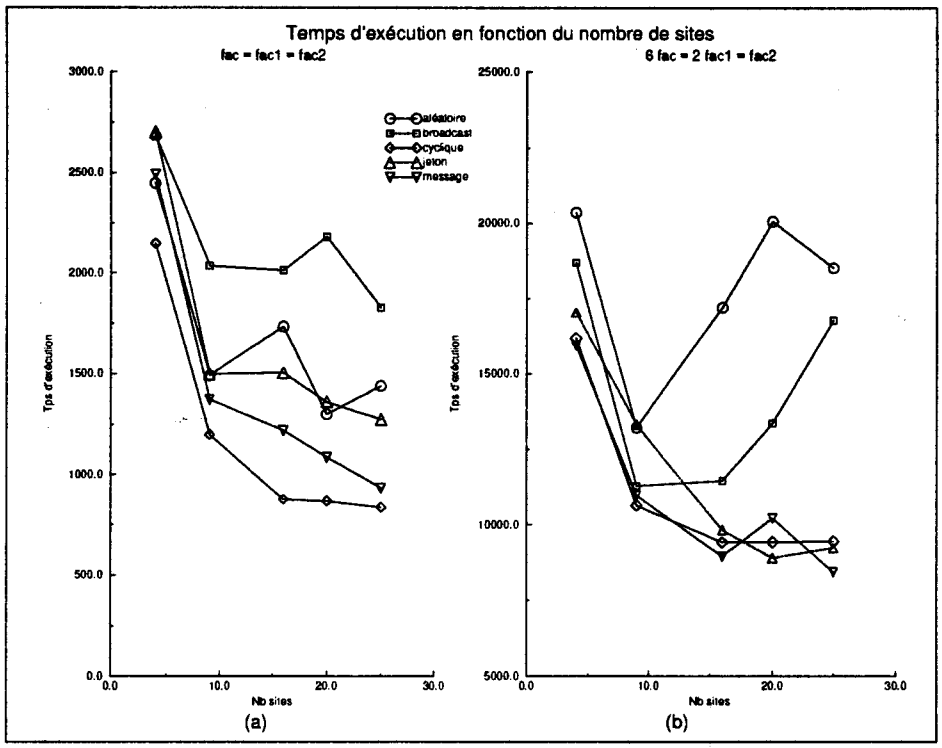


FIG. 2.11 - *Résultats d'exécution pour l'application factorielle*

comportements dans les modules, nous préparons le travail de répartition dynamique de charge. Pour cela nous dupliquons le plus possible, dans la limite des contraintes de la machine et des spécifications du programmeur, le code de chacun des comportements. Les possibilités de régulation sont dictées par le nombre de fois qu'aura été dupliqué un comportement. Le choix d'un noeud pour l'exécution d'un comportement se limite aux sites ayant chargé un module contenant son code.

Le taux de création dynamique de processus, l'évolution dynamique du graphe de relation entre les processus, et une variation de la complexité de chaque comportement d'une application nous amènent à proposer au programmeur une *collection* de stratégies de répartition dynamique de charge qui ont comme caractéristiques communes d'utiliser un algorithme de décision décentralisé, une duplication de l'état global de la machine, et de ne pas permettre la remise en cause du placement d'un processus au cours de son exécution. Les stratégies proposées s'articulent autour des quatre composantes qui sont : 1) une politique de calcul d'un indicateur de charge; 2) une politique de transfert de charge; 3) une politique d'information; 4) une politique de localisation.

L'outil de répartition proposé demande de la part de l'utilisateur de fournir une instanciation des paramètres. Pour cela nous avons développé une plate-forme de simulation permettant de faire un réglage des différents paramètres en fonction

de l'application et de l'architecture de la machine, mais aussi d'analyser l'exécution, et de déterminer les caractéristiques relatives de chaque comportement d'une application. La présentation de cette plate-forme fera l'objet du chapitre 3.

Chapitre 3

Méthode d'évaluation des performances du placement des Cacs

3.1 Modélisation et évaluation des systèmes et des applications

Comme nous l'avons vu dans le chapitre précédent, le problème du placement des Cacs ne peut pas être abordé classiquement. Les algorithmes de placement statique qui utilisent uniquement les informations obtenues grâce à l'analyse du source de l'application et en affectant un coût à chaque tâche de l'application ne permettent pas de prendre en compte le caractère dynamique que constitue le modèle d'exécution décrit par les Cacs. D'un autre côté, les algorithmes de placement dynamique qui se basent en général sur un état global de la machine parallèle, ne tiennent pas compte des caractéristiques de l'entité à placer, ce qui peut aboutir à des placements qui n'exploitent pas toutes les possibilités de la machine, et qui peuvent même conduire à une augmentation du temps d'exécution.

Le nombre de paramètres à étudier pour gérer le placement des Cacs nous a conduit à développer un outil nous permettant dans un premier temps de les identifier, pour ensuite évaluer leurs incidences sur le comportement de l'exécution d'une application. Cet outil consiste en une modélisation de l'ensemble des architectures de machines parallèles utilisées pour exécuter le modèle Cac, et une modélisation du comportement des Cacs constituant une application. A l'aide de

cette modélisation nous avons développé un simulateur nous permettant d'étudier le comportement des architectures et des applications modélisées.

3.1.1 Les différentes techniques de modélisation de systèmes informatiques

La modélisation dans un cadre général consiste à trouver une correspondance entre le problème étudié et un autre problème équivalent dont on sait parfaitement identifier et analyser le comportement. En ce qui concerne la modélisation de systèmes informatiques, le problème équivalent généralement utilisé est l'étude d'un réseau de files d'attente [All80]. Chaque composante du système est modélisée par un objet *station* dans la terminologie QNAP2 [Sim88], une **station** se caractérise par les éléments suivants :

- une *file d'attente*,
- un ou plusieurs *serveurs* attachés à cette file,
- la description algorithmique du *service*,
- un algorithme de routage qui indique le déplacement des *clients* à travers le réseau.

Les **clients** se déplacent entre les stations pour demander un **service** à l'un des **serveurs**.

La **file d'attente** est caractérisée par :

- A: la suite des instants d'arrivée des clients dans la file, encore appelée processus d'entrée (M pour loi exponentielle, D pour loi constante, ...),
- S: la durée des services, ou processus de sortie,
- C: le nombre de serveurs,
- K: la capacité maximale de la file,
- L: la population des usagers,
- DS: la discipline de service (Premier arrivé, premier servi; Dernier arrivé, premier servi; Aléatoire; ...)

On rencontre généralement la notation de Kendall pour désigner ses caractéristiques : A/S/C (DS/K/L) ou A/S/C/K/L (DS) ou encore A/S/C/K/L/DS. Quand les trois derniers éléments ne sont pas spécifiés K et L ont pour valeur

l'infini et la discipline de service est premier arrivé premier servi (par exemple M/M/1 définit une file d'attente avec des entrées et des sorties qui suivent une loi exponentielle, avec un seul serveur et les autres valeurs par défaut).

Cette modélisation est généralement utilisée pour quantifier les performances d'un système en donnant le débit, le temps de réponse, les ressources utilisées, et le temps de leur utilisation.

Une autre modélisation basée sur les réseaux de Petri permet de contrôler la validité du comportement du système. Un réseau de Petri est défini par :

- Des places qui représentent les états des ressources du système;
- Des transactions qui représentent les opérations réalisées, ou les informations de contrôle;
- Des arcs reliant les places aux transactions qui représentent l'exécution des opérations.

Les places peuvent être marquées. On définit un marquage initial, ce marquage évolue en fonction de régles de franchissement de transactions. L'étude des différents marquages accessibles à partir du marquage initial permettent de faire l'analyse comportementale du système.

Les précédentes modélisations sont bien adaptées aux systèmes simples, cependant dès que le système devient plus important, il faut faire de nombreuses approximations pour conserver les hypothèses de l'outil de modélisation qui éloigne le modèle du système réel. Quand on veut modéliser plus finement le comportement de chaque composante d'un système, on est amené à le modéliser en donnant l'algorithme de chaque élément.

On trouvera dans [FP89] une présentation des différents techniques de modélisation de systèmes informatiques.

3.1.2 Les différentes techniques de modélisation d'applications

Les différentes modélisations de systèmes informatiques présentées précédements nécessitent la description des lois suivies par les entrées des clients (tâches ou processus). Soit on ne les connaît pas, dans ce cas on utilise un générateur d'événements suivant une loi aléatoire comme la loi exponentielle, loi de poisson, etc.... Une autre possibilité consiste à enregistrer une trace d'exécution sur le système réel (ce qui suppose de disposer du système réel, et de perturber le moins possible l'exécution par enregistrement de la trace). Une fois la trace obtenue, c'est elle qui cadence les entrées dans la modélisation du système.

Dans notre cas les événements sont générés par l'exécution d'une application suivant le modèle défini dans le projet PVC/BOX. Nous avons choisi de créer le langage *GENESE* pour **GENE**rateur d'une **S**uite d'**E**vénements qui modélise une exécution en générant la suite d'événements caractéristique de l'application. Cette suite est ensuite envoyée comme entrée dans la modélisation de notre système informatique (Voir paragraphe 3.2.3).

Lorsque l'on veut étudier le comportement en terme de performance de l'exécution d'une application, la modélisation peut être une solution qui permet de faciliter sa mise en place, de réduire le temps de développement et d'évaluation. Certaines de ces modélisations ont pour but de montrer le bon fonctionnement d'un programme. Cette optique utilisée dans le cadre du développement d'outil de déverminage [Paz89] sort du contexte de notre étude.

Le langage *ANDES* dans le projet *ALPES* [KP92, KTP93] permet de décrire le comportement d'applications parallèles qui suivent le modèle d'exécution CSP [Hoa78], c'est à dire des applications composés de tâches qui communiquent par envoi de messages synchrones.

L'exécution peut être représentée par un graphe orienté de tâches, où les sommets sont les tâches et les arcs, les relations de précédences et éventuellement les relations de communications. S'il y a un arc entre une tâche a et une tâche b, alors la tâche b ne sera exécutée au plus tôt que quand la tâche a aura terminé son exécution suivie éventuellement d'un envoi de message vers b.

Une tâche est définie par :

- les entrées,
- un coût de calcul et de communication éventuel,
- les sorties.

Les entrées (sorties) implémentent un mécanisme de synchronisation du type *And/Or*. Pour les entrées, l'opérateur *And* entre plusieurs entrées signifie que l'exécution ne peut commencer que lorsque toutes les entrées sont disponibles, l'opérateur *Or* implémente le comportement du *Alt* dans le langage *OCCAM*. Pour les sorties l'opérateur *And* signifie que le message est envoyé à l'ensemble des destinataires. Différentes variantes de l'opérateur *Or* permettent de choisir de un à plusieurs destinataires.

Les coûts de calcul et de communication permettent de faire des évaluations de performance sur l'exécution de l'application parallèle modélisée.

Le projet Paralex [BAAD91] permet de décrire des applications dont l'exécution est représentée par un graphe de type *data-flow* simplifié, les tâches sont des fonctions écrites dans un langage séquentiel (langage C). Le projet Visage

[RZZ93] met en place un environnement de programmation pour applications parallèles modélisées par des graphes.

3.1.3 Evaluation des systèmes informatiques

Une fois que la modélisation du système informatique est définie, il faut mettre en place son évaluation. Il faut alors définir les grandeurs que l'on veut mesurer comme le taux d'occupation de la CPU, le taux d'occupation de la mémoire, le nombre de processus en attente de traitement, ...

Différentes techniques d'évaluation sont présentées dans la suite, elles ont leurs avantages et leurs inconvénients, cependant on peut affirmer que l'évaluation ne peut apporter que des tendances, et que l'erreur par rapport à la réalité dépend à la fois du modèle choisi (qui peut être plus ou moins précis) et de son évaluation (qui perturbe plus ou moins le cours normal d'une exécution).

3.1.3.1 Evaluation analytique

Dans de nombreux travaux, par exemple [ELZ86, KL87, GLR84], les auteurs utilisent une évaluation analytique basée sur un réseau de files d'attente pour évaluer des stratégies d'équilibrage de charge. Cette évaluation a l'intêret de pouvoir être résolue mathématiquement, donc rapidement par des méthodes d'optimisation. Cependant les types de modélisations permettant cette évaluation sont réduits, et reposent sur de nombreuses simplifications du système de départ qui peut amener à des résultats de performances estimées éloignés des performances réelles du système.

3.1.3.2 La simulation

La simulation est utilisée pour évaluer un système dont le modèle ne correspond plus aux contraintes d'une évaluation analytique.

On rencontre deux types de simulation. L'une est basée sur l'utilisation d'une génération d'événements discrets, c'est à dire que l'on génère des événements qui suivent des lois aléatoires. Cette simulation utilise en général un modèle basé sur un réseau de files d'attente. L'utilisateur spécifie les lois aléatoires qu'il souhaite en entrée.

L'autre est basée sur l'utilisation d'une trace enregistrée sur la machine réelle. Dans ce cas, il faut s'assurer que la trace est représentative vis à vis de l'évaluation des performances que l'on veut en tirer. Le modèle du système est alors rarement basé sur un réseau de files d'attente, mais plus généralement représenté par un algorithme qui décrit son comportement.

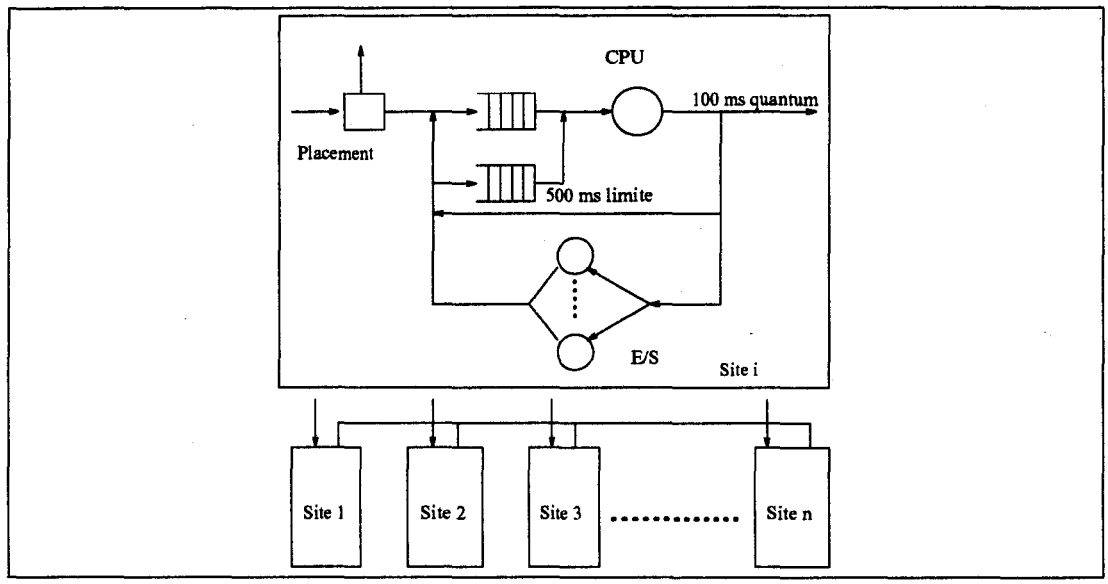


FIG. 3.1 - *Structure du modèle utilisé pour la simulation de Zhou*

Zhou [Zho88] utilise l'enregistrement d'une trace du comportement d'un réseau de stations de travail pendant une durée assez longue. Cette trace contient :

- La date de démarrage d'une tâche,
- la charge CPU qu'elle représente,
- le nombre d'entrées/sorties disque effectuées pendant son exécution.

Il utilise ces traces pour évaluer différentes stratégies d'équilibrage de charge sur le modèle d'exécution de la figure 3.1.

3.1.4 Evaluation des applications modélisées

L'évaluation des applications peut se faire sur une architecture de machine parallèle (ou un réseau de stations de travail comme pour le projet Paralex). On obtient alors une évaluation réelle des performances de l'application. Dans le cas où l'évaluation se fait sur une machine qui émule le comportement d'une machine parallèle, les exécutions sont comparées en faisant varier les paramètres de la machine parallèle émulée ou les stratégies d'exécution (La machine parallèle Meganode pour le projet *ALPES* émule différentes architectures dont les caractéristiques sont spécifiées dans un fichier de paramètres). Les différentes exécutions sont alors exploitées par des outils graphiques qui visualisent les données à évaluer comme la charge CPU, le trafic en communication et l'occupation mémoire. Ce

travail s'effectue soit en parallèle avec l'exécution, avec l'inconvénient de perturber fortement l'exécution, soit en *post-mortem* à l'aide d'une trace obtenue par des routines chargées de piéger les événements importants au cours de l'exécution.

3.2 Présentation de notre plateforme d'évaluation

3.2.1 Les objectifs

Comme nous l'avons vu (voir présentation du contexte de l'étude, Chapitre 2), le placement des Cacs nécessite une adaptation des stratégies de placement en fonction du type de Cacs et de l'architecture sur laquelle va s'exécuter l'application. La plateforme d'évaluation développée doit fournir une solution en aidant le programmeur à choisir, entre les différentes possibilités qui lui sont données, les bons paramètres des stratégies de placement. Ce problème de placement ne pouvant être dissocié du problème du découpage de l'application pour augmenter les performances d'une application, la plateforme d'évaluation doit donner au programmeur la possibilité d'étudier le comportement d'une application en faisant varier son découpage.

La plateforme d'évaluation a donc les objectifs suivants :

- Permettre une évaluation des stratégies de placement

  Nous avons développé un ensemble de stratégies de placement qui nous paraissent caractéristiques et qui répondent le mieux aux types d'applications et d'architectures auxquelles elles vont être soumises. L'utilisateur aura à déterminer avant le lancement de la simulation, les stratégies de placement qu'il veut utiliser. Pour chacune des stratégies, il disposera d'une gamme de paramètres lui permettant de régler finement son comportement. A la suite d'une simulation, des résultats, sous forme de traces et de tableaux en forme de compte-rendus, fourniront autant d'éléments de comparaison lui permettant d'évaluer le comportement de son application pour l'architecture donnée et l'ensemble des stratégies de placement sélectionnées.

- Déterminer les caractéristiques des entités utilisées

  La plateforme doit recueillir des informations sur le coût d'exécution des entités, et les relations de communication entre les entités. Ces informations seront réinjectées dans le code de description de l'application pour permettre à l'algorithme de placement de prendre en compte les caractéristiques de l'entité et ainsi faciliter leur placement. Le même type d'informa-

tion peut être ajouté dans le source de l'application réelle lors de la phase d'exploitation.

- Aider au découpage de l'application

  La plateforme doit permettre d'établir un découpage de l'application en entités. Grâce à un examen des traces de simulation, on pourra établir une liste des points sensibles d'une application, les goulots d'étranglement, un découpage trop gros qui entraîne une attente du reste des entités, ou un découpage trop fin qui génère des communications trop coûteuses.

- Déterminer la topologie adaptée

  La plateforme doit permettre de tester un ensemble d'architectures de machines. Celles-ci ayant les caractéristiques communes suivantes :

  - Un réseau de communication
  - Un ensemble de sites avec des ressources particulières, et des puissances variables. Chaque site dispose d'une mémoire locale privée.

  L'ensemble des topologies testées permettront de déterminer celle qui est la plus adaptée à l'application.

3.2.2 Architecture générale

La figure 3.2 donne le synoptique du fonctionnement de la plateforme d'évaluation. En entrée l'utilisateur décrit l'architecture qu'il veut utiliser en donnant la liste des sites et une description du réseau de communication inter sites. Il fournit la description de son application dans le langage GENESE. Enfin il spécifie la ou les stratégies de placement qu'il veut tester en indiquant pour chacune d'elle les paramètres de réglage.

L'ensemble de ces données est fourni en entrée du simulateur qui traite les événements générés par l'interprétation des instructions fournies par le langage de description de l'application. Le simulateur traite les événements en fonction de l'architecture décrite et de la stratégie de placement selectionnée. L'utilisateur peut visualiser la répartition dynamiquement grâce à une fenêtre de visualisation.

Chaque simulation produit une trace qui peut être visualisée et fournit des informations en forme de compte rendu de simulation.

De l'ensemble des informations obtenues, on peut déterminer la complexité de chaque entité, et un graphe de relations entre les entités. Sur l'ensemble des simulations on peut obtenir le paramétrage de la meilleure stratégie de placement pour l'application, et l'architecture utilisée.

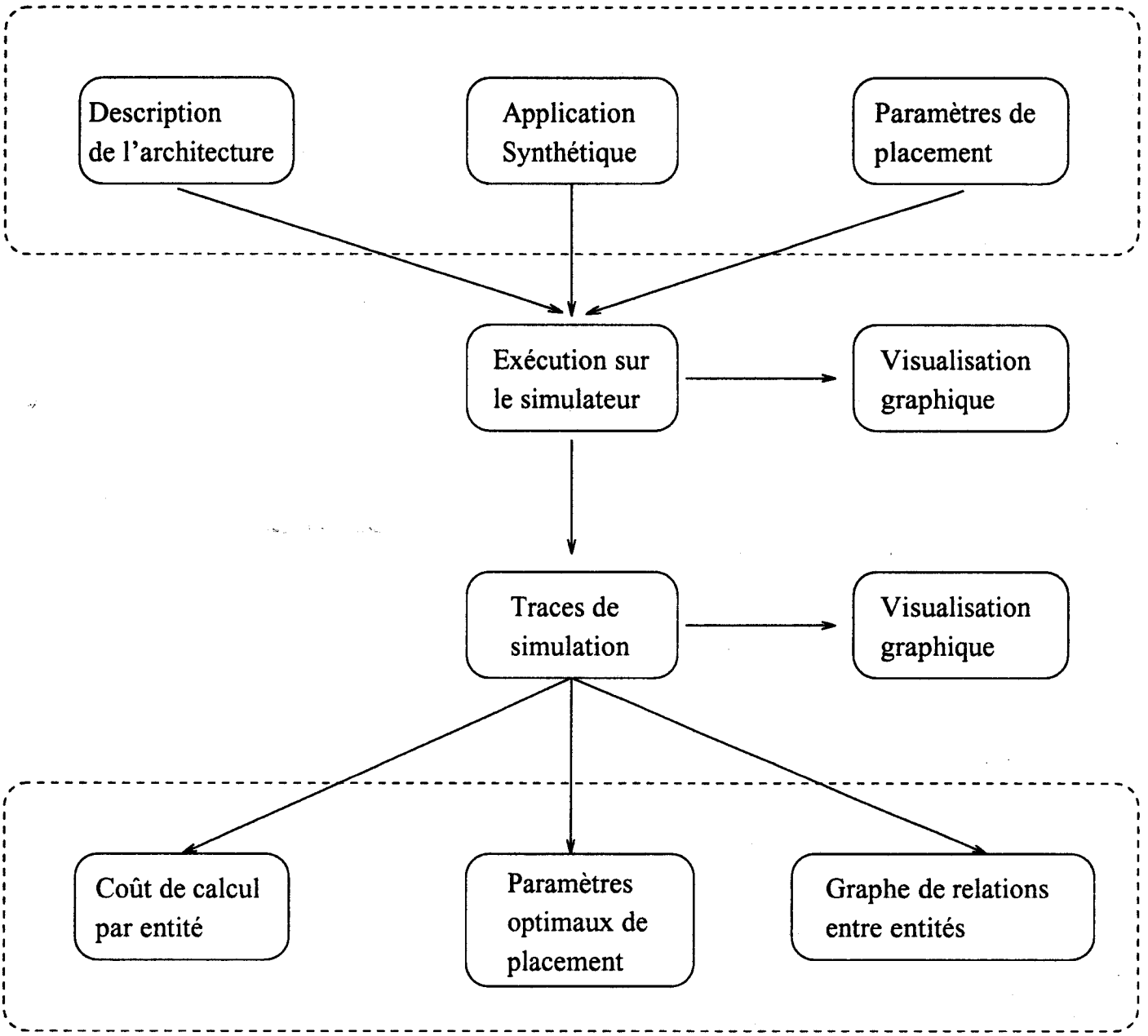


FIG. 3.2 - *Synoptique de la plateforme d'évaluation*

3.2.3 L'application en langage GENESE

3.2.3.1 Modélisation des applications

Pour répondre aux objectifs fixés par la plateforme, il nous fallait un moyen de fournir en entrée de notre simulateur une suite d'événements correspondant à l'exécution d'une application composée de Cacs. Pour cela nous avions la possibilité d'enregistrer la trace d'une exécution réelle en utilisant la réalisation du run-time développé sur une machine parallèle [CRGM92], comme dans l'étude réalisée par Zhou [Zho88]. Cependant il nous fallait pouvoir ajouter dans ces traces les informations obtenues lors d'une précédente simulation. Cela afin d'évaluer l'apport des informations sur les stratégies de placement, et de permettre une modification de la description de l'architecture (voir fig 3.3).

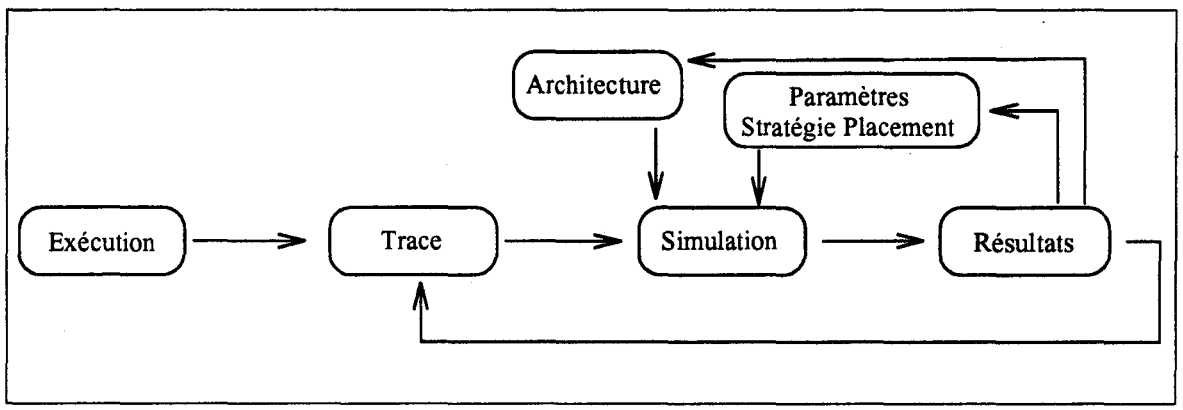


FIG. 3.3 - *Trace d'une exécution réelle*

Nous avons préféré développer le langage GENESE qui à partir d'une description simplifiée d'une application composée de Cacs, permet de générer une trace d'événements représentative d'une exécution réelle. Le langage permet facilement d'intégrer les informations fournies par les précédentes simulations. De plus, le langage utilise une syntaxe assez simple qui nous permet d'étudier le comportement d'une application en modifiant facilement son découpage (voir fig 3.4).

3.2.3.2 Le langage de description de l'application GENESE

Le langage que nous avons développé a pour but de rendre possible la description d'une application sous forme d'un ensemble d'entités actives communicantes. Les entités décrites ont les mêmes propriétes que les Cacs du projet PVC/BOX. On y retrouve une activité correspondant au processus décrit dans la définition du Cac, des données locales constituant un environnement privé, et enfin un nom unique pouvant être utilisé comme champ destination lors d'un échange de

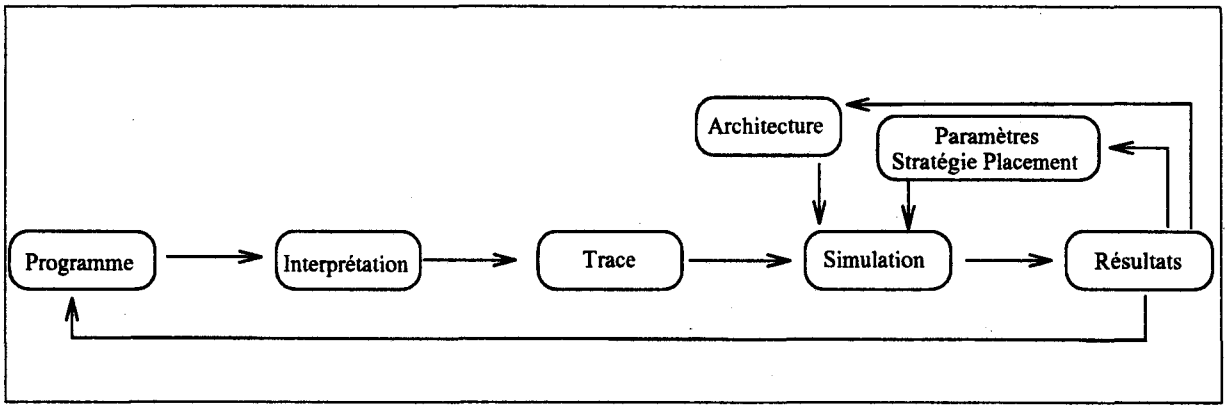


FIG. 3.4 - *Trace provenant d'une interprétation du langage GENESE*

message, c'est à dire la boîte aux lettres d'un Cac. La description des entités se décompose en deux parties, la première donne une description de l'environnement dans lequel le programmeur souhaite que l'entité se trouve lors de son exécution, la deuxième permet de donner une description synthétique de son comportement.

Description de l'environnement d'une entité

La description de l'environnement d'une entité a pour but de faciliter le travail de l'algorithme de placement dynamique en ajoutant des informations lui permettant de choisir plus precisément un site lors d'une création. Ces informations sont de plusieurs natures et de plusieurs provenances :

- Informations obtenues dans les étapes antérieures du développement de l'application
  - La taille mémoire que représente l'occupation de l'entité pendant son exécution obtenue lors de l'analyse du source de l'application. Un cac étant le regroupement d'une boîte aux lettres et des données utilisés par le comportement qu'il exécute, cette taille est fonction des données (attributs) locales.
  - La liste des ressources nécessaires à son exécution obtenue aussi dans la phase de l'analyse du source de l'application, comme un écran graphique, un capteur dans une application temps réel, etc ....
  - La liste des sites susceptibles d'accueillir l'entité obtenue à la suite de la phase du regroupement du code des entités en modules utilisés au chargement de l'application.

    Remarque : Ce regroupement peut conduire à une limitation dans le choix d'un site destination lors d'une demande de création.

- Informations obtenues dans les simulations précédentes
  - La liste des liaisons avec les autres entités avec pour chaque liaison un coefficient permettant de valuer la liaison ou évaluer l'importance de la liaison en terme de volume, nombre d'échanges, ou en nombre de créations effectuées (qui représente une liaison de dépendance).
  - Le temps moyen consacré par la CPU pour exécuter chaque instance d'une entité. On considère que chaque instance consomme le même temps CPU.
  - Une moyenne de la durée de présence de chaque instance d'une entité pendant l'exécution de l'application.
  - Le nombre moyen d'attente de messages pour chaque instance d'une entité.
- Les volontés exprimées par le programmeur de l'application.

  Ces volontés permettent au programmeur de spécifier des relations de localisation entre les entités que ne peuvent pas être obtenues automatiquement, mais dont le programmeur a constaté l'importance lors de précédentes exécutions. Il est à noter, que si ces informations peuvent être bénéfiques pour la répartition, elles peuvent aussi avoir un effet désastreux sur le temps d'exécution. Cette phase d'analyse du comportement de l'exécution permet aussi de constater les répercutions de ces informations sur l'exécution.
  - Exprimer un nombre maximum d'exécutions simultanées d'instances d'une même entité sur un site.
  - Donner des dépendances de localité, qui peuvent exprimer des contraintes de sécurité, ou des contraintes de tolérance aux pannes.
- Contraintes exprimées par l'utilisateur

  Les contraintes que peut exprimer l'utilisateur sont de deux natures :
  - Les contraintes matérielles

    Les contraintes matérielles regroupent toutes les demandes particulières de ressources nécessaires pour l'exécution de l'entité (par exemple un disque local, un écran graphique, ...).
  - Les contraintes de localisation

    On trouve deux types de contraintes de localisation. Celles qui définissent un *domaine* dans lequel une entité va pouvoir s'exécuter. Ce *domaine* est défini statiquement par l'utilisateur, il permet de dédier une partie de son architecture à un ensemble d'entités. Dans la figure 3.5, `entite_1` se voit affecter les sites `0, 2, 4, 6, 8, 10, 12, 14`.

```
root()
{
--
--  Specification
--
  taille := 5000;
  domaine := 0;
    .
    .
--
--  comportement
--
  begin
    .
    .
  end
}
entite_2(...)
{
  taille := 1000;
  domaine := 1, 3, 5, 7, 9, 11,
             13, 15;
  sans := entite_3;
    .
    .
  begin
    .
    .
  end
}
```

```
entite_1(...)
{
  taille := 1000;
  domaine := 0, 2, 4, 6, 8, 10,
             12, 14;
  necessite := "Disque";
  avec := entite_3;
  sans := entite_2;
    .
    .
  begin
    .
    .
  end
}


entite_3(...)
{
  taille := 1000;
  nb_max := 10;
    .
    .
  begin
    .
    .
  end
}
```

FIG. 3.5 - *Exemples de spécifications de contraintes dans une entité*

Les contraintes dynamiques au contraire s'appliquent en fonction du placement des autres entités déja créées.

- **avec** ***nom_entité***: Indique l'obligation de créer l'entité sur un site contenant l'entité *nom_entité.*
- **sans** ***nom_entité***: Indique l'obligation de créer l'entité sur un site ne contenant pas l'entité *nom_entité.*
- **nb_max** ***nb_entité***: Indique l'obligation de créer l'entité sur un site ne contenant pas plus de *nb_entité* entité(s).

Remarques Le fait de pouvoir mettre des contraintes sur le choix des sites possibles pour la création d'une entité est une chose, pouvoir respecter ces contraintes en est une autre. En effet pour que ces contraintes soient garanties, il faut que l'information dont on dispose soit bonne. Cela veut dire qu'il faut choisir une stratégie d'échanges d'information qui centralise les informations et un algorithme de décision qui soit lui aussi centralisé. Dans ce contexte les contraintes statiques sont satisfaites quelque soit la politique d'information utilisée, par contre les contraintes dynamiques peuvent être vérifiées sur un site au regard des informations de charge disponibles et se révéler violées dans la réalité si on utilise un autre type de stratégie d'échanges d'information et un autre type d'algorithme de décision que le centralisé. La volonté d'obligation apparaît donc plutôt comme un souhait exprimé par l'utilisateur, si on veut pouvoir utiliser d'autres solutions que le centralisé, peu adapté à notre contexte d'étude.

Description du comportement d'une entité

Dans la description synthétique du comportement, on doit retrouver les instructions importantes du modèle d'exécution PVC/BOX. Les instructions utilisées pour décrire le comportement d'une entité sont celles qui ont une incidence sur l'exécution. Nous avons retenu les instructions suivantes :

- **CREATION** *creation [(num)] nom_entité (par_creat_entité)*

  L'instruction *creation* permet de créer dynamiquement une nouvelle entité *nom_entité* avec les paramètres de création *par_creat_entité.* Le paramètre optionnel *num* permet au programmeur de choisir un numéro de site sans laisser choisir l'algorithme de placement. L'instruction *creation* renvoie le numéro d'identification unique de l'instance de l'entité créée. Ce numéro sera utilisé pour désigner l'entité.

- **MESSAGE** *message [coût] nom_destinataire pars_message*

  L'instruction *message* permet d'envoyer un message vers *nom_destinataire* contenant les données *pars_message*, le paramètre optionnel *coût* permet de

modifier le coût facturé, ceci pour prendre en compte des tailles variables de messages. la variable *nom_destinataire* est le numéro d'identification de l'entité destinataire. L'envoi de message n'est pas bloquant, le mécanisme de communication est donc asynchrone.

- **ATTENTE** *attente [exp] pars_message*

  L'instruction *attente* permet de bloquer une entité en attente d'un message, dont les données seront rangées dans *pars_message*. Le paramètre optionnel *exp* permet de filtrer les messages en provenance d'un expéditeur particulier. la variable *exp* correspond au numéro d'identification de l'entité emettrice du message.

  Une communication synchrone est facilement obtenue en combinant une instruction envoi de message *message* suivi immédiatement par une instruction *attente*.

- **CALCUL** *calcul [coût]*

  L'instruction *calcul* permet d'exprimer une consommation CPU en unités de temps logique. Le paramètre optionnel permet de faire varier la valeur du coût par défaut. Elle permet de cacher la complexité d'un bloc d'instructions, autre que Création, Message et Attente, et dont l'analyse n'a pas d'intérêt direct pour aider à l'étude de l'application. Cette instruction permet de découper plus ou moins finement la description d'une application, mais permet aussi de localiser à l'intérieur du déroulement d'un comportement les parties représentant une charge de calcul.

Ces instructions ont comme point commun de permettre de tracer les évènements importants pour l'étude du comportement d'une application, mais aussi de permettre l'étude du placement dynamique des entités.

Une instruction **Fin** a été ajoutée pour faciliter la gestion des entités lors de leur exécution sur le simulateur.

Les autres instructions telles que les instructions de branchement (si, tantque) et les instructions d'affectation ne génèrent pas d'événements. Elles permettent de décrire le comportement de l'entité réelle.

Contrairement à certaines études sur le comportement d'applications dans l'environnement d'une architecture distribuée [CA82, ELZ86, KP92], il était important pour nous d'assurer un parfait déterminisme dans le séquencement de l'exécution des entités d'une simulation à une autre. Ceci afin de pouvoir comparer les temps logiques d'exécution en faisant varier par exemple les paramètres des stratégies de placement. C'est pourquoi les tests des contrôles de séquencement dépendent essentiellement de données fournies au lancement de la simulation d'exécution de l'application, et ne reposent pas sur des lois probabilistes.

Exemple: Le crible d'Eratosthène

L'application est représentée comme une collection d'entités qui synthétisent le comportement des Cacs. L'entité particulière Root est utilisée pour lancer l'application. L'exemple du crible d'Eratostène permet d'obtenir les nombres premiers compris entre 2 et une borne N fixée par l'utilisateur. L'implantation parallèle qui en a été faite (voir le code en annexe B.1), repose sur l'idée suivante [Noa92] : chaque nombre premier constitue une entité, l'ensemble des entités sont reliées pour former un *pipeline* des nombres premiers rangés par ordre croissant (voir figure 3.6), il donne un aperçu des possibilités du langage. L'application est composée de deux entités :

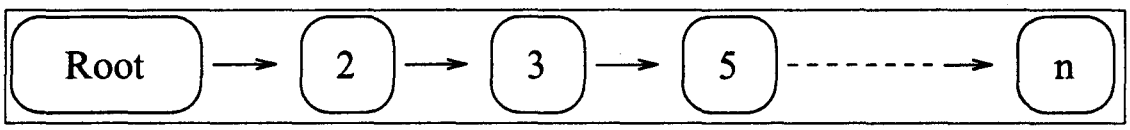


FIG. 3.6 - *Le crible d'Eratosthène*

L'entité *root* initialise le calcul, demande la création de l'entité *eratos* contenant le nombre premier *"2"*. A la suite de quoi, une boucle d'envoi de message contenant un nombre entier, alimente en données à traiter l'entité précédemment créée.

L'entité *eratos* qui contient un nombre premier p, fait la division du nombre entier reçu par message par la valeur p. Si le reste est non nul, alors l'entier est envoyé vers l'entité suivante dans le *pipeline*, sinon il ne fait rien et attend le message suivant. Dans le cas où le reste n'est pas nul mais que l'entité est la dernière dans le *pipeline*, cela signifie que l'entier reçu est un nombre premier. Il est ajouté en queue du *pipeline* en faisant une demande de création d'une nouvelle entité.

La figure 3.7 donne un extrait du code de l'entité `eratos`. Les lignes 1-7 permettent de définir l'environnement d'exécution de l'entité. Les lignes 9-11 représentent l'initialisation de variables locales à l'entité. La ligne 14 bloque l'activité dans l'attente d'un message sans préciser sa provenance. L'instruction `calcul` de la ligne 19 provoque un événement qui sera traité par le simulateur, il traduit le coût de calcul des lignes 17-18. La ligne 22 provoque la création d'une nouvelle activité de l'entité `eratos`.

```
1 eratos(racine,pere,valeur)
2 {
3   taille := 1000;
4   type := CEM;
5
6   var donnee, rqt, next, fini, q, r,cal;
7
8   begin
9     cal := 100;
10    fini := 0;
11    next := -1;
12    tantque (fini == 0) faire
13    begin
14      attente (donnee, rqt);
15      if (rqt == -1) then
16      begin
17          q := donnee / valeur;
18          r := donnee - (q * valeur);
19          calcul [cal];
20          if (r != 0) then
21            if ( next == -1) then
22              next := creation eratos (racine,self, donnee);
23            else
24              message next (donnee, rqt);
25      end;
26      else  -- rqt == -2
27      begin
28        if (next == -1) then
29          message racine ();
30        else
31          message next (donnee, rqt);
32        fini := 1;
33      end;
34    end;
35    fin;
36  end
37 };
```

FIG. 3.7 - *Extrait du programme du crible d'Eratostène*

3.2.4 Modélisation du système informatique

3.2.4.1 Justification du choix de notre modélisation

La modélisation de la gamme de systèmes informatiques qui nous intéresse (c'est à dire une collection de sites connectés par un réseau de communication sans mémoire commune), ne permettait pas d'en faire une évaluation analytique théorique [FP89] étant donné le nombre important de sites et des mécanismes de dépendance entre sites à mettre en place. La simulation restait la seule solution permettant de pouvoir évaluer facilement plusieurs architectures en modifiant les paramètres les caractérisant, et en évitant un coût de développement important pour chacune d'elle. La mise en place de cette simulation pouvait difficilement être obtenue en utilisant un réseau de files d'attente pour les raisons suivantes :

- La complexité de la description du comportement d'un site en terme de files d'attente.
- La possibilité de vouloir évaluer l'exécution d'une application dans un environnement hétérogène aurait multiplié la complexité par autant d'architectures utilisées dans le réseau.
- La difficulté d'exprimer la loi aléatoire suivie par le processus d'entrée des files d'attente vis à vis du modèle d'exécution particulier utilisé dans notre projet.

3.2.4.2 Description d'un site

Un site dans notre simulation (voir figure 3.8) est modélisé par une unité de traitement (CPU) qui incrémente un compteur, représentant une horloge locale (HORLOGE), à chaque fois qu'elle exécute une instruction qui génère un événement pour le compte d'un processus (PROCESSUS). Chaque site reçoit les messages à destination des processus qui s'exécutent sur le site dans une boîte aux lettres unique (BOX). Nous allons présenter chaque élément du site en détail.

3.2.4.2.1 Les processus. On distingue trois types de processus :

- Le processus prioritaire;

  Le processus prioritaire est créé au lancement de la simulation indépendement de l'application. Le processus prioritaire est celui qui va exécuter, quand c'est nécessaire, le code permettant d'implémenter la stratégie de placement dynamique. C'est lui qui est chargé de mettre en place la stratégie d'échanges d'informations, et l'algorithme de localisation.

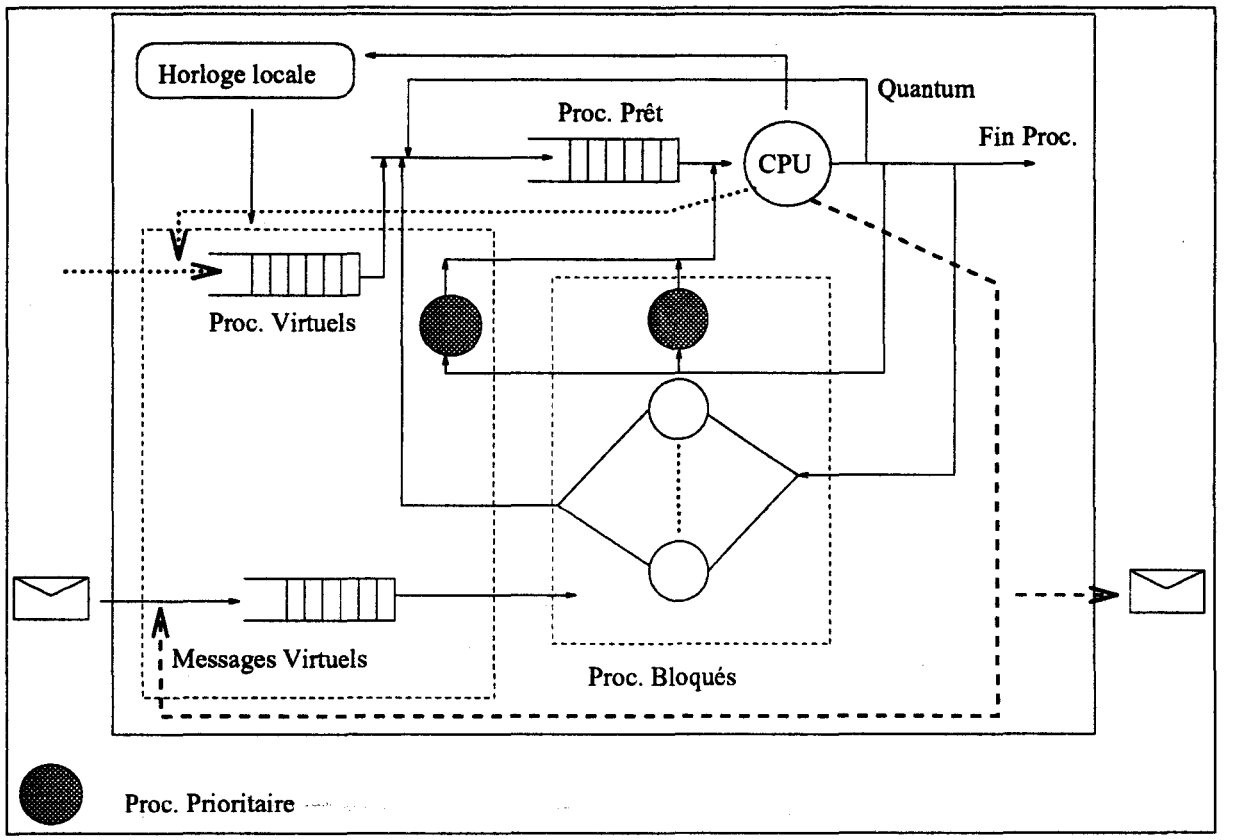


FIG. 3.8 - *Modélisation des machines parallèles cibles*

Les processus créés dynamiquement à la suite de l'exécution d'une instruction *création.*

- Les processus prêts;

  Un processus prêt représente l'activité de l'instance d'une entité. Il exécute le code qui décrit son comportement dans le programme défini par l'utilisateur, les instructions génératrices d'événements comme les autres (affectation, tests et branchement). Seules les instructions tracées "consomment du temps CPU" et par voie de conséquence font incrémenter le compteur de l'horloge. Les processus prêts sont rangés dans une liste qui est utilisée par la CPU pour déterminer celui qui sera exécuté. Cette liste est parcourue dans un ordre *premier arrivé, premier servi.*

- Les processus virtuels.

  Les processus virtuels sont des processus qui du fait de la simulation ont été créés, mais n'existent pas encore réellement. Ces processus seront considérés comme prêts dès que l'horloge locale aura une valeur supérieure ou égale à la valeur de la date de création du processus.

3.2.4.2.2 La CPU. La CPU exécute les instructions du processus prioritaire s'il est prêt ou sinon du premier processus prêt dans la liste. Le processus prioritaire est exécuté tant qu'il reste actif. Les autres processus sont exécutés par *quatum* de temps logique. Après chaque *quatum*, si le processus est toujours actif il est rangé en fin de liste des processus prêts. L'algorithme utilisé par la CPU pour le scheduling est décrit dans la figure 3.9.

```
proc
    tq (1) faire
      si Etat(Proc_Prio) = Prêt alors
        tq Etat(Proc_Prio) = Prêt faire
          execute_instruction(Proc_Prio)
        fait
      sinon
          si Non_Vide_Liste(Proc_Act) alors
            tps ← 0
            tq (tps < Quantum) ∧ (Etat(Premier_Liste(Proc_Act)) = Prêt) faire
              tps ← tps + execute_instruction(Premier_Liste(Proc_Act))
            fait
            P ← Premier_Liste(Proc_Act)
            Retire_Premier_Liste(Proc_Act)
            si Etat(Premier_Liste(Proc_Act)) = Prêt alors
              Insert_Queue_Liste(Proc_Act, P)
            sinon
                  Insert_Liste(Proc_Bloq)
            fsi
          fsi
      fsi
    fait
```

FIG. 3.9 - *Algorithme de scheduling utilisé par la CPU*

3.2.4.2.3 L'horloge. Chaque site dispose d'une horloge locale. Cette horloge locale implémentée par un compteur est modifiée dans deux circonstances :

- Soit un processus prêt (prioritaire ou non) a exécuté une instruction tracée, dans ce cas on incrémente le compteur par la valeur de facturation pour cette instruction.
- Soit aucun processus n'est prêt (prioritaire ou non) alors dans ce cas on incrémente le compteur jusqu'à la date du premier événement possible.

Cette incrémentation correspond à un espace de temps où le site est "*idle*", c'est à dire qu'il n'a rien à faire.

3.2.4.3 Calcul du coût d'exécution des instructions tracées.

Coût de communication entre deux sites distants Le coût de communication entre deux sites quelconques i et j est égal au coût de communication pour une liaison multipliée par le nombre de liaisons parcourues pour aller d'un site i au site j (et inversement) et cela en choisissant toujours le plus court chemin. On ne tient pas compte des interférences dûes aux surcharges du trafic, ni d'un algorithme de routage dynamique. Ce coût unitaire entre deux sites est fourni comme paramètre de l'architecture simulée. L'utilisateur peut augmenter ce coût en donnant un facteur multiplicateur simulant des tailles de messages variables (voir l'instruction message).

Le coût de création. Le coût de Création représente la réservation de l'espace mémoire nécessaire à l'exécution d'une entité, et sa prise en compte par le système [HLDM91]. Ce coût est éclaté pour modéliser au mieux le comportement du run-time PVC (Voir Fig. 3.10). Il se décompose comme suit :

Lors de l'exécution d'une demande de création, cette demande déclenche localement l'algorithme de détermination du site destinataire. Une fois déterminé, un message de demande de création est envoyé au site qui a été choisi pour recevoir l'entité à créer. Après l'envoi du message le processus se bloque en attente du retour du numéro d'identification de l'entité créée.

Sur le site destinataire il y a création d'une référence qui est envoyée au processus demandeur, puis création effective du processus dans un état *virtuel* si l'horloge locale a une valeur inférieure à la date de la mise en service du processus, ou dans un état *Actif* dans le cas contraire.

Le coût d'une création pour la CPU locale au site demandeur correspond au temps de recherche du site destinataire, suivi d'un envoi de message, et d'une réception de message. Pour le site destinataire cela correspond à une reception de message, un envoi de message, et la création d'un environnement pour un processus (réservation de mémoire). Le coût de création est donné comme paramètre de l'architecture simulée. Ce coût est réparti de la manière suivante : 10% pour la détermination du site destinataire et 90% pour la création de l'environnement du processus sur le site destinataire suite à des mesures effectuées sur les architectures où le run-time PVC a été implantée. Le coût des envois et des réceptions de messages étant fournis par ailleurs.

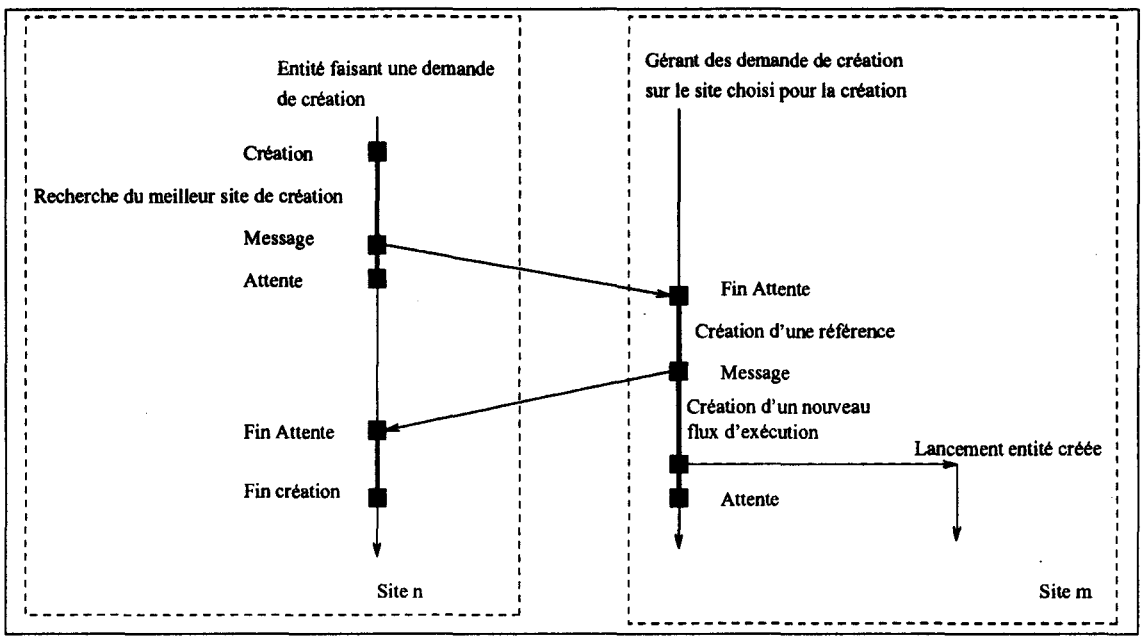


FIG. 3.10 - *Création d'un nouveau flux d'exécution*

Le coût d'un envoi de message. Le coût de l'envoi d'un message englobe la mise en forme du message et sa prise en charge par le système de communication du site.

Le coût d'une réception de message. Le coût de Attente représente le traitement à la réception du message sur le site et le temps de lecture par l'entité.

Le coût de calcul. Le coût de Calcul permet de relativiser les instructions précédement indiquées par rapport aux instructions plus classiques de calcul, d'affectation, ou encore de branchement.

3.2.4.4 Fonctionnement d'un site

Au lancement de la simulation, chaque site est en attente d'une instruction à exécuter. Un processus (prioritaire ou non) prêt va fournir les instructions à exécuter. La CPU applique sur le processus prêt l'algorithme de scheduling décrit dans l'algorithme figure 3.9. Le processus prêt peut être bloqué dans l'attente de réception d'un message, dans ce cas il est mis dans la liste des processus bloqués.

Dans le cas où le processus exécute une instruction d'envoi de message, le message est mis dans la liste de messages du site contenant le destinataire du message. Les messages sont rangés par ordre croissant de leur date de validité. En effet un délai de communication entre sites est simulé en ajoutant un temps

en unités logiques au temps du site contenant l'émetteur du message (ce délai correspond au coût de communication entre deux sites présenté plus haut). Le message est valide pour le site destinataire uniquement si le temps local en unités logiques est supérieur ou égal à sa date de validité. Si le site contenant l'émetteur est le même que celui contenant le destinataire du message, le temps de communication est considéré comme nul. Si la liste des messages contient un message valide et qu'il y a un destinataire dans la liste des processus bloqués qui l'attend, alors ce processus est rangé en queue de la file d'attente des processus prêts.

Dans le cas où le processus exécute une instruction de création d'un nouveau processus, l'algorithme décrivant la stratégie de placement est déclenchée, celui-ci fournit le numéro du site qu'il a estimé comme le meilleur pour prendre en charge l'exécution. Si le numéro du site élu est égal au numéro du site ayant fait la demande, alors le nouveau processus est mis en queue de la file des processus prêts. Si le numéro du site est différent du site local on range le nouveau processus dans la liste des processus virtuels du site élu. Cette liste est rangée par ordre croissant de date de mise en service des processus virtuels. Cette date correspond au temps en unités logiques de l'horloge du site ayant fait la demande de création auquel on a ajouté le temps de communication (en unités logiques) pour atteindre le site élu pour l'exécuter. Le processus virtuel est mis en queue de la file des processus prêts uniquement quand sa date de mise en service est inférieure ou égale à l'horloge du site contenant la liste. La liste des messages et la liste des processus virtuels sont parcourues à chaque changement de contexte de processus.

Détermination de la date du premier événement possible. Lorsque le site est "idle", c'est à dire qu'aucun processus n'est prêt, on incrémente le compteur jusqu'à la date du premier événement possible.

Ces événements sont le déblocage d'un processus en attente d'un message présent mais pas encore valide vis à vis de la valeur de l'horloge locale, ou le démarrage d'un processus virtuel.

Dans un premier temps on détermine l'événement futur le plus proche sur le site.

- Dans la liste des messages "*virtuels*": on compare les dates de validité et on choisit la valeur la plus petite,
- Dans la liste des processus "*virtuels*": on compare les dates de mise en service, et on choisit la valeur la plus petite.

L'événement le plus proche correspond à la valeur minimale entre les deux dates trouvées. Si la liste des messages *virtuels* et la liste des processus *virtuels* sont vides, alors la date du prochain événement a une valeur infinie.

Pour garantir qu'aucun autre événement extérieur (création d'un nouveau processus, envoi d'un message) n'ait une date antérieure à celle trouvée, on consulte les horloges des autres sites.

Si toutes les horloges ont une valeur supérieure à l'horloge du site considéré, alors la date trouvée est la bonne. On peut donc incrémenter la valeur de l'horloge jusqu'à cette valeur, et débloquer un processus. Dans le cas contraire, on ne peut pas garantir que le site ayant une valeur d'horloge inférieure ne va pas produire un événement sur le site examiné avec une date antérieure à la date du prochain événement possible trouvé. Dans ce cas on choisit d'avancer l'horloge locale jusqu'à la valeur de l'horloge du site qui a une valeur minimale, il en est de même quand aucune date pour le prochain événement n'a été trouvée.

3.2.4.5 Fonctionnement du système modélisé

Pour simuler le comportement d'une machine parallèle sur une station de travail, il faut s'arranger pour que chaque site évolue de façon quasi simultanée. Pour cela le mécanisme de sélection d'un des sites se présente de la manière suivante :

1° On compare l'horloge de chaque site, et on choisit celui qui a une valeur d'horloge inférieure à toutes les autres.

2° Pour ce site on examine la liste des processus prêts. Si la liste n'est pas vide, alors on exécute les instructions jusqu'à épuisement du quantum ou jusqu'à blocage du processus. Si la liste des processus prêts est vide, alors on avance l'horloge comme indiqué dans le paragraphe "*Détermination de la date du premier événement possible*"

3° A la suite de quoi, on recommence le traitement 1.

Ce mécanisme global de fonctionnement associé à l'évolution de l'horloge et du traitement des processus par la CPU, permet d'effectuer un enchainement des événements cohérent avec une des possibilités d'exécution sur une machine parallèle (à l'indéterminisme prêt).

Notre souci est ici de s'approcher de la situation réelle et donc de garantir que les traces obtenues par la simulation pourraient avoir été obtenues sur une machine parallèle.

3.2.5 La description de l'architecture

La description de l'architecture de la machine parallèle précise les particularités de l'architecture choisie dans l'ensemble des machines visées par le projet PVC/BOX : les multicomputers. Ceux-ci ont pour caractéristique globale d'avoir

un modèle d'exécution MIMD, chaque site est composé d'une mémoire locale, d'un processeur, d'un système de communication rapide, et d'un ensemble de ressources particulières. Les informations de la mémoire ne sont pas accessibles par les autres sites. Les sites communiquent par envoi de messages sur un réseau de communication (voir fig 3.11).

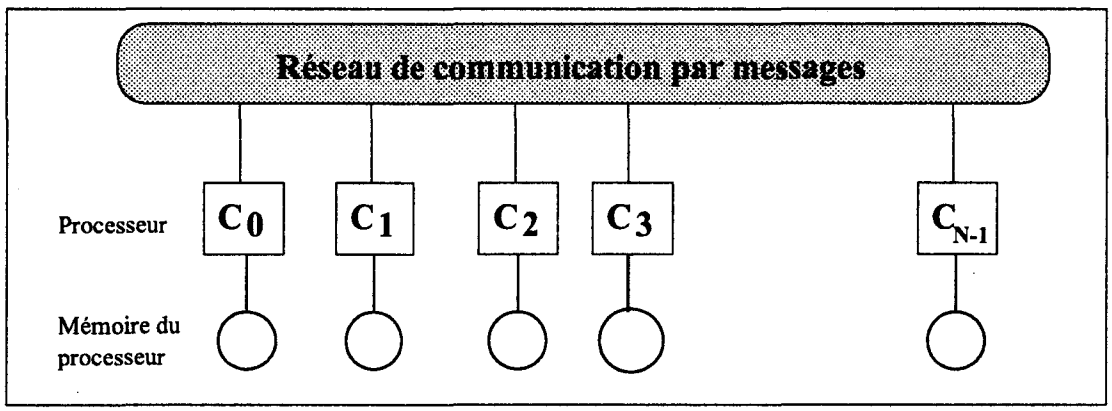


FIG. 3.11 - *Architecture d'un multicomputer*

La description de l'architecture, permet d'une part de spécifier pour chaque site ses particularités et d'autre part de définir la topologie du réseau en spécifiant les liaisons présentes entre les sites.

La description d'un site

Par ce dispositif nous avons la possibilité de définir les caractéristiques importantes pour chaque site au regard des problèmes que nous étudions, c'est à dire leur incidence sur le placement. Ces caractéristiques sont les suivantes (voir Fig. 3.12) :

- la mémoire disponible,
- la puissance relative du processeur du site par rapport aux autres, et enfin
- les ressources particulières présentes sur le site.

Cette description permet de simuler des architectures hétérogènes, notamment en relativisant la puissance des machines les unes par rapport aux autres.

La description des liaisons entre les sites et les paramètres globaux de l'architecture

Les liaisons sont décrites point à point (voir Fig. 3.13). S'il n'y a pas de liaison entre un site i et un site j, un message échangé entre ces deux sites sera routé sur

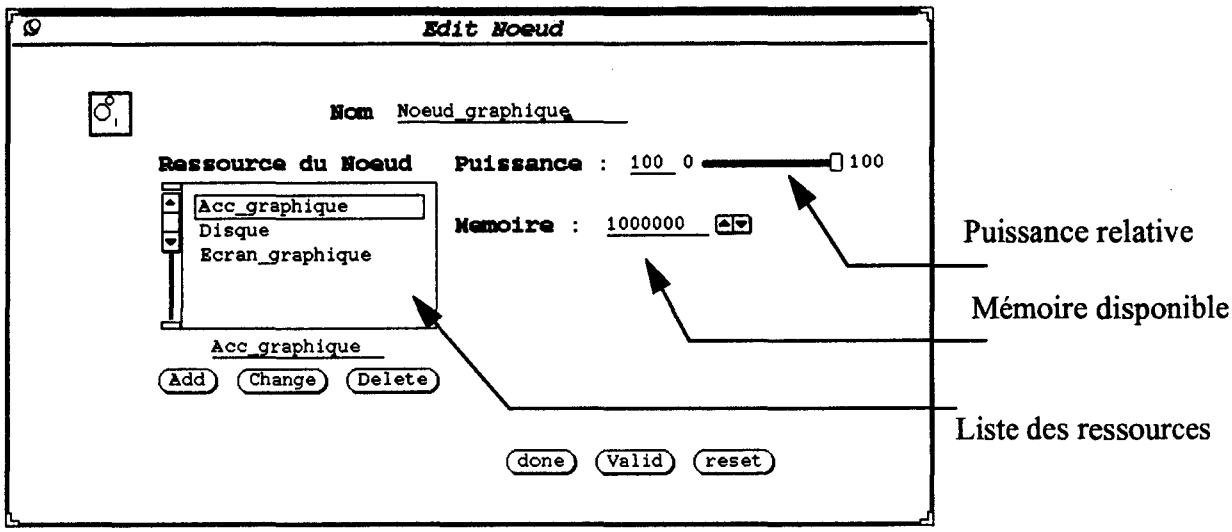


FIG. 3.12 - *Edition des caractéristiques d'un site*

le chemin le plus court possible entre les deux sites. Dans notre étude il n'a pas été tenu compte du problème de routage dynamique, ni du partage de la liaison par la transmission de plusieurs messages simultanément.

Nous avons ensuite ajouté des paramètres globaux à l'architecture qui vont préciser le type d'architecture. Ces paramètres représentent les coûts unitaires en unité de temps logique des instructions tracées, c'est à dire **Calcul, Création, Attente, Message** et le **coût de communication** entre deux sites voisins. Cela nous permet de préciser le coût de l'un par rapport aux autres, et d'analyser le comportement des stratégies de placement en fonction de ces variations.

3.2.6 Les paramètres de stratégie de placement

L'intérêt de cette plateforme d'évaluation est de pouvoir faire évoluer les stratégies de placement testées. L'ensemble des stratégies de placement proposées ont comme point commun les caractéristiques suivantes :

- Disposer d'un algorithme de décision distribué, chaque site est autonome dans le choix du placement des entités à créer. Un algorithme de décision centralisé n'est pas adapté à notre modèle d'exécution, où le nombre de demandes de création simultanées peut être important, et introduirait un goulot d'étranglement dans le déroulement de l'exécution. Si on prend comme exemple d'application le calcul de factorielle n présenté dans le pa-

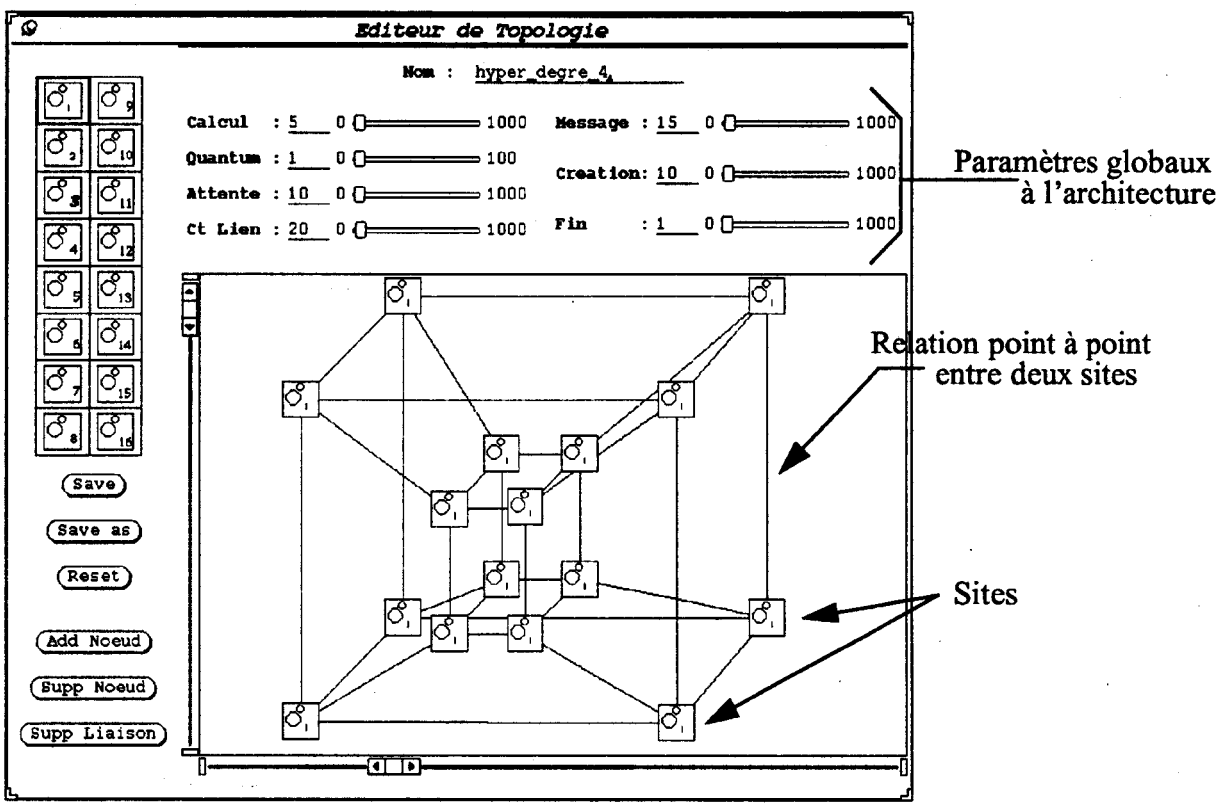


FIG. 3.13 - *Edition de l'architecture de la machine simulée*

ragraphe 2.3.2.4, son exécution va provoquer la demande de création de $2n-1$ entités *fac*, le nombre de demandes simultanées augmente avec n, l'utilisation d'un algorithme de décision centralisé augmenterait les temps de réponse et le temps d'exécution global de l'application.

- Le choix du site où s'exécute une entité n'est pas remis en cause au cours de son exécution, autrement dit il n'est pas envisagé de mécanisme de migration de tâches. Ce choix repose sur le fait que dans un premier temps l'architecture de la machine parallèle envisagée pour implémenter le runtime PVC/BOX (MultiCluster II de chez Parsytec) ne permettait pas de le mettre en place (le transputer constituant un site de la machine ne le permet pas). De plus il s'avère que le temps moyen d'exécution d'une entité est inférieur au temps que représente la gestion d'une migration c'est à dire:
  - le temps de transfert du contexte de l'entité,
  - la gestion de la redirection des messages,
  - le sur-coût de communication.

Cependant sans qu'il n'ait été étudié, le mécanisme de migration permet, quand il est réalisable sur la machine cible, d'enrichir les possibilités de régulation et/ou d'équilibrage de charge dans le cas d'un placement dynamique. La frontière de rentabilité d'un tel mécanisme peut être dépassée lorsque l'exécution de certaines entités provoque un sur-coût de communication, ou encore un déséquilibre de charge prolongé [BSS91, JV89].

Les caractéristiques des stratégies de placement se divisent en quatre composantes qui ont été présentées dans le chapitre 1.

- La méthode d'évaluation de la charge d'un site
- La politique d'information
- La politique de transfert
- La politique de localisation

3.2.6.1 Intégration des stratégies de placement dans la simulation

Les stratégies de placement dynamique sont intégrées dans le simulateur de façon à pouvoir évaluer leurs coûts et les perturbations qu'elles entrainent sur l'exécution d'une application.

Utilisation du panneau de configuration

L'utilisateur définit les stratégies de placement qu'il veut évaluer pour son application (voir Fig. 3.14). Pour cela il choisit une ou plusieurs stratégies d'échanges d'informations, l'indicateur de charge qu'il va échanger grâce à la (ou aux) stratégie(s) choisie(s), la stratégie de transfert, et enfin une ou plusieurs stratégies de localisation. Certaines stratégies peuvent être paramétrées, ces paramètres seront présentés ultérieurement.

Le processus prioritaire

La plupart des stratégies d'échanges d'informations demande la création d'un processus permettant sa mise en place. Ce processus décrit un algorithme appelé $lb(i)$, *lb* pour load balancing, et le paramètre i pour le numéro du site sur lequel il est exécuté. L'algorithme est exécuté par le processus prioritaire décrit dans le paragraphe 3.2.4.2.1.

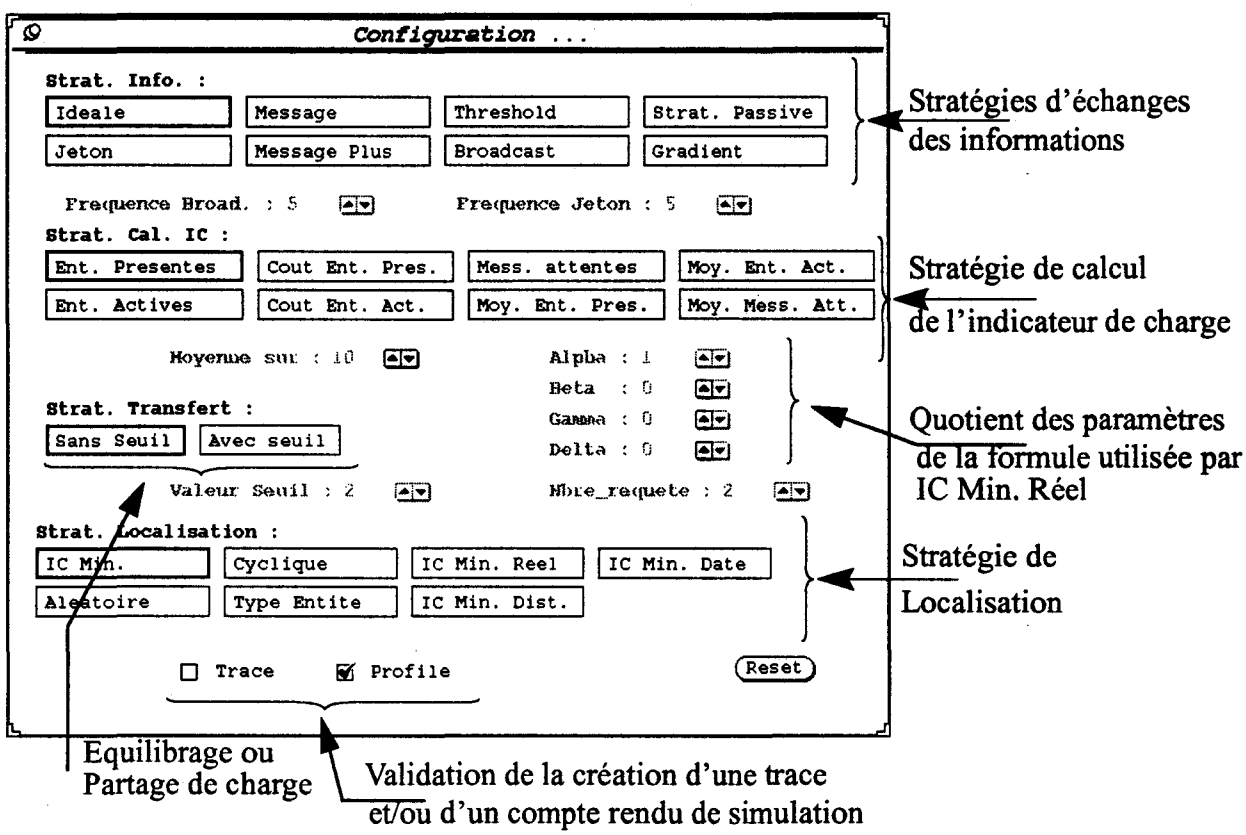


FIG. 3.14 - *Ecran de configuration des stratégies de placement*

Les structures de données

Comme l'ensemble des stratégies de placement ont un algorithme de localisation et un mécanisme d'échanges d'information décentralisés, chaque site dispose d'un état plus ou moins global de la machine. La structure de donnée tableau *tab_IC* contenant les informations de charge de l'ensemble des sites est présente sur chaque site. Chaque site dispose des paramètres qui ont été positionnés par l'utilisateur sur le panneau de configuration, ils seront présentés lorsqu'ils seront utilisés.

La variable *Nb_sites* qui indique le nombre de sites présents dans l'architecture est obtenue en utilisant la description qui en a été faite lors de sa configuration (voir paragraphe 3.2.5).

Les fonctions

Le processus prioritaire utilise les mêmes instructions tracées qu'un processus classique, mais ici adaptées pour faciliter leur utilisation. Des fonctions supplémentaires ont dû être implémentées.

- *Dormir(Période)* qui bloque le processus pendant une certaine période.
- *Valeur_IC()* qui donne la valeur de l'indicateur de charge en fonction de l'état du site local.
- *message_pour_lb()* qui retourne un booléen avec la valeur vraie si il y a un message pour le processus prioritaire *lb*() et faux sinon.
- *lire_message_lb()* qui lit le premier message pour $lb(i)$.

3.2.6.2 La méthode d'évaluation de la charge d'un site

Ce paramètre est celui qui guide le choix de la stratégie de placement. Il doit permettre de caractériser ce qui représente une charge vis à vis de l'architecture et vis à vis du type d'exécution de l'application. Avec une architecture comme le MultiCluster II, où chaque site dispose d'une mémoire réduite (2Mo), il faut tenir compte de son occupation mémoire pour caractériser la charge et éviter un défaut mémoire à l'exécution d'un processus. Par contre, l'occupation mémoire d'une station de travail qui dispose d'un mécanisme de mémoire virtuelle a moins d'importance pour caractériser sa charge.

Une application qui serait composée d'entités toutes identiques en terme de coût d'exécution ou en terme de coût de communication, n'a pas besoin d'utiliser un indicateur de charge qui prendra en compte ces informations. Dans ce cas un compteur d'entités est représentatif. Par contre quand les entités sont très différentes en terme de coût, le calcul de la charge ne peut plus se contenter d'un compteur simple, il faut pouvoir pondérer ce compteur par le type de l'entité.

L'intérêt d'un indicateur de charge c'est aussi de pouvoir être un indicateur de tendance de charge, même si dans notre cas la migration n'est pas envisagée, et que le problème d'instabilité du système pour cause de déplacements intempestifs d'entités n'existe pas. En effet la valeur qui va être diffusée sur les autres sites doit pouvoir garder une certaine vraisemblance pendant le temps qui sépare deux échantillonnages. Les indicateurs de charge selectionnables sont les suivants :

- Les entités présentes

  Dans ce cas, on ne fait pas de distinctions entre les entités qui sont en attente d'exécution (*Prêtes* et celles qui sont bloquées en attente d'un message). La

valeur renvoyée par *valeur_IC()* est le nombre total de processus présents sur le site. C'est à dire ceux qui sont dans la liste des processus prêts et ceux qui sont dans la liste des processus bloqués.

- Les entités prêtes

  Cet indicateur de charge est généralement utilisé sur les stations de travail, avec des processus à gros grain, c'est à dire qui ont une durée de vie qui peut être importante, mais qui utilisent peu ou moyennement la CPU. La charge réelle dans ce cas est mieux représentée par les processus prêts [FZ86]. La valeur renvoyée par *valeur_IC()* est le nombre total de processus présents dans la liste des processus prêts.

- Les entités présentes pondérées par leur coût respectif

  Cet indicateur de charge nécessite qu'une estimation du coût de l'entité soit connue au moment de l'exécution. Cette information peut provenir d'une analyse du source, ou obtenue par une précédente exécution. Dans le contexte de notre étude le fait de pouvoir inclure une pondération par type d'entité est un plus pour certaines applications. Dans le cadre de la simulation le coût est un coefficient permettant de pondérer les entités les unes par rapport aux autres. Par exemple, si lors d'une précédente simulation pour une application contenant deux entités A et B, le coût d'exécution de l'entité A valait 218 et qu'il valait 120 pour l'entité B alors, à chaque création d'une entité A la valeur de l'indicateur de charge d'un site sera incrémentée deux fois plus que pour l'entité B.

- Le nombre de messages en attente

  On peut considérer, dans un modèle d'exécution où toutes les entités communiquent par échanges de messages, que plus un site a une liste de messages importante, plus sa charge de travail future est importante, ou encore que plus la liste de messages est importante et plus le temps de réponse pour chacun d'eux sera important. Donc, mieux seront répartis les messages à traiter, et plus court sera le temps de réponse, meilleur sera le temps global d'exécution de l'application. La valeur renvoyée par *valeur_IC()* est ici le nombre total de messages en attente de traitement.

- Moyenne sur l'évolution de la charge calculée avec l'une des méthodes d'évaluation précédentes.

  Dans un souci de disposer d'un indicateur de charge qui ne tienne pas uniquement compte de l'instant de l'évaluation de la charge, les indicateurs de charge présentés précédemment peuvent être ajustés par un calcul d'une moyenne sur un certain nombre de valeurs enregistrées. La période d'échantillonnage et le nombre de valeurs devant permettre de faire disparaître les

valeurs non significatives, et ne garder que les tendances. Le paramètre introduit par le label "Moyenne sur:" donne la possibilité de faire varier la période d'échantillonnage pour calculer la valeur moyenne.

Dans le contexte du modèle d'exécution développé par le run-time PVC, l'occupation mémoire d'un site est fonction du nombre de Cacs présents sur ce site. Lors de la création d'un cac on lui affecte un volume mémoire qui est fixé par le run-time. Dans ces conditions l'indicateur de charge représenté par le nombre de Cacs présents donne ici de plus une bonne évaluation de l'occupation mémoire.

3.2.6.3 La politique d'information

Les différentes politiques d'informations proposées, se différencient par le nombre de sites joints lors de la diffusion d'une modification de la charge d'un site, et par la méthode de diffusion. La politique d'information représente la plus grosse perturbation introduite par la stratégie de placement dynamique. Son choix est intimement lié aux possibilités de l'architecture, mais aussi à la variation de la charge produite par l'application. Quand la politique d'échanges d'informations le nécessitera, un processus prioritaire $lb(i)$ sera implémenté pour gérer le mécanisme sur chaque site i.

Anticipation sur la modification des indicateurs de charge

Le taux de création des entités peut être important notamment au lancement de l'application, dans ces conditions la charge des sites augmente rapidement sans qu'il soit possible à un quelconque mécanisme d'échanges d'informations de véhiculer ces modifications. C'est pourquoi la valeur de l'indicateur de charge du site j élu lors d'une demande de création sur un site i est modifiée par anticipation. Ceci afin que les informations disponibles sur le site i concernant le site j soient plus justes lors des demandes de création futures, surtout si le site i fait de nombreuses demandes de création.

Dans le simulateur, l'information de charge visible pour le site j disponible sur le site i est modifiée dès que le site j a été choisi, et cela quelque soit la politique d'information choisie

Nous avons implémenté diverses politiques d'échanges d'informations, qui se distinguent par le mode de diffusion.

3.2.6.3.1 Diffusion globale: BROADCAST L'indicateur de charge est diffusé vers l'ensemble des autres sites à chaque changement représentatif.

Seuil est la valeur qui définit le changement représentatif de l'indicateur de charge. L'échantillonnage de l'indicateur de charge est fait périodiquement. La variable *Période* est utilisée pour régler la période d'échantillonnage. La mise en place de ce mécanisme est réalisée en endormant la tâche prioritaire $lb(i)$. La fréquence de mise à jour des indicateurs de charge est fonction de ces deux valeurs. Les paramètres *Seuil* et *Période* peuvent être modifiés par l'intermédiaire du panneau de configuration.

```
proc lb(i)
    old_IC ← 0
    Initialise_Seuil(Seuil)              - Récupère les valeurs définies par
    Initialise_Periode(Periode)                          - l'utilisateur
    tq (1) faire
      tq message_pour_lb() faire                  - Indicateur de charge en
        lire_message_lb(IC, n)                  - provenance des autres sites
        tab_IC(n) ← IC
      fait
      IC ← valeur_IC()
      si ||old_IC − IC|| > Seuil alors
        diffuse_nouveau_IC()                    - diffusion de la nouvelle valeur
        old_IC ← IC
      fsi                                                - de l'indicateur de charge
      dormir(Periode)
    fait
```

La fonction *diffuse_nouveau_IC()* a été implémentée par une boucle d'envoi de message vers l'ensemble des autres sites. Cela, étant donné que la possibilité de diffuser un message complétement (*Broadcast*) ou vers un sous ensemble de destinataires (*Multicast*) en une seule opération n'est pas présente sur l'ensemble des architectures.

3.2.6.3.2 Diffusion locale : JETON Dans le cas du jeton, les informations concernant la charge des sites sont rangées dans un vecteur. Le *Jeton* circule de site en site. Le site qui contient le *Jeton*, prend en compte les nouvelles valeurs de la charge des autres sites et modifie la valeur correspondant à son indicateur de charge dans le vecteur, après quoi, il passe le *Jeton* au suivant.

La variable *Periode* permet de réguler la fréquence de passage du *Jeton*. Une modification de cette valeur dynamiquement permet d'adapter la politique d'information à l'évolution de la charge.

```
proc lb(i)
    Initialise_Periode(Periode)
    tq (1) faire
        attente_message (IC[1..Nb_sites])                    - Attente du Jeton
        pour j = 1 → Nb_sites faire
            si j ≠ i alors
                tab_IC(n) ← IC[j]
            fsi
        fait
        IC[i] ← valeur_IC()                                  - Modification de l'IC du site
                                                               - dans le jeton
        dormir(Periode)
                                                             - Envoi du Jeton
        envoi_message lb(i + 1Modulo Nb_sites) (IC[1..Nb_site])
    fait
```

La méthode du gradient qui est à diffusion locale également fera l'objet d'un paragraphe particulier, étant donné qu'elle implique la politique de tranfert, la politique de localisation, et la politique d'information. Dans ce cas, l'état de la charge est diffusé de proche en proche par un effet de propagation (voir paragraphe 3.2.6.6).

3.2.6.3.3 Diffusion par l'application : MESSAGE Les informations relatives à la charge sont échangées en utilisant les messages de communication entre les entités et les demandes de création distante (voir fig. 3.15). Cette politique d'information n'ajoute aucun *overhead* de communication. La validité des informations de charge dépend de la fréquence des échanges entre les entités. Lors d'un envoi de message, la procédure de mise en forme du message ajoute dans l'entête les informations de charge. Donc à chaque envoi d'un message d'un site A vers un site B, l'information de charge de A sera connue sur le site B. En cas de création, comme il y a un échange de messages, s'il y a une demande de création d'un site A vers un site C, alors le site A reçoit un état de la charge du site C et le site C reçoit un état de la charge du site A.

3.2.6.3.4 Diffusion à la demande: THRESHOLD et MESSAGE PLUS La politique d'information *Threshold* (fig. 3.16) fait une demande d'information de charge vers *Nb_requêtes* sites choisis cycliquement à chaque demande de création d'une entité. Une fois les demandes d'informations faites, la politique de localisation s'applique sur l'ensemble des sites sans attendre les réponses, c'est une manière de ré-actualiser les informations de charge sur un sous-ensemble des

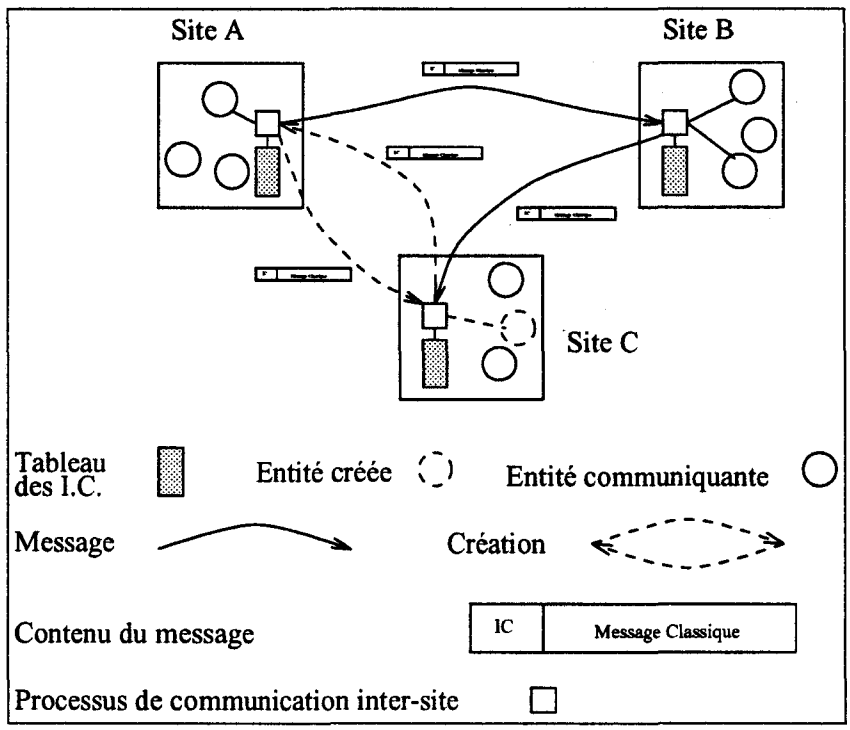


FIG. 3.15 - *Echanges des indicateurs de charge avec la politique d'information MESSAGE*

sites. Dans le contexte de notre étude étant donné que le taux de création est important, toute attente pour une création pénalise le temps global d'exécution. C'est pourquoi on n'attend pas que les demandes d'informations soient revenues pour faire le choix du meilleur site. La variable *Nb_requêtes* peut être modifiée par l'utilisateur par l'intermédiaire du panneau de configuration.

Message Plus est une politique qui permet en partie de répondre à l'inconvénient de la méthode *Message* qui ne dispose d'une information de charge valide que pour les sites avec lesquels il communique beaucoup. En plus de l'indicateur de charge de sites, on conserve la date de sa dernière mise à jour. Lors d'une demande de création, s'il y a des informations dont on considère que la période de validité est dépassée, on applique la même technique que pour la politique d'information *Threshold*. C'est à dire que l'on fait une demande de mise à jour des informations de charge vers *Nb_requêtes* sites choisis cycliquement.

3.2.6.3.5 STRATPASS La stratégie STRATPASS met en place une stratégie où l'échange d'informations est à l'initialive des sites qui sont oisifs (ils demandent du travail aux sites chargés, voir fig. 3.17), contrairement à toutes les autres stratégies présentées jusqu'à présent. Comme indiqué dans le chapitre 1, cette politique d'information s'applique plutôt quand un mécanisme de migration

```
proc lb(i)
    next_site ← 0
    tq (1) faire
        attente_message (rqt, n, IC)
        choix rqt faire
            cas Demande :
                pour j ← 0 :→ Nb_requêtes faire
                    si next_site = i alors
                        next_site ← (next_site + 1) Modulo Nb_sites
                        envoyer_message lb(next_site) (Question, i, 0)
                    sinon
                        envoyer_message lb(next_site) (Question, i, 0)
                        next_site ← (next_site + 1) Modulo Nb_sites
                    fsi
                fait
            cas Question :
                IC ← Valeur_IC()
                envoyer_message lb(n) (Réponse, i, IC)
            cas Réponse :
                tab_IC(n) ← IC
        fchoix
    fait
```

FIG. 3.16 - *Algorithme d'échange des indicateurs de charge pour THRESHOLD et MESSAGE PLUS*

```
proc lb(i)
    old_IC ← 0
    tq (1) faire
        tq message_pour_lb() faire
            lire_message_lb(n, IC)
            tab_IC(n) ← IC
        fait
        IC ← valeur_IC()
        si (Old_IC ≠ IC) ∧ (IC < Seuil) alors
            si (Old_IC ≥ Seuil) alors Diffuse_voisin(i, IC) fsi
            Old_IC ← IC
        fsi
        Dormir(Période)
    fait
```

FIG. 3.17 - *Algorithme de diffusion des indicateurs de charge pour STRATPASS*

est implémenté, car la demande des sites oisifs permet de décharger les sites surchargés. Cependant, lorsque les entités sont peu coûteuses en terme de charge, mais nombreuses, cette stratégie peut permettre de faciliter le choix mis en place dans la politique de localisation.

Dans un premier temps l'algorithme teste sa boîte aux lettres pour savoir s'il a reçu des demandes de travail de la part de ses voisins. Ensuite, on regarde l'état de la charge locale. Si la charge est inférieure à un certain seuil *Seuil* alors on demande du travail aux voisins en diffusant sa valeur d'indicateur de charge. Quand on analyse l'algorithme, on s'aperçoit qu'un site diffuse son indicateur de charge uniquement lorsqu'il cherche du travail. Il se peut donc qu'un site après avoir demandé du travail soit à son tour surchargé. Une solution consiste à inclure dans l'algorithme de localisation la prise en compte de la date de dernière modification de l'indicateur de charge pour un site. Plus l'information est vieille pour un site, moins il faut en tenir compte. On considère alors que le site est devenu surchargé, et on choisit un site dont on a une information de charge plus récente. La variable *Seuil* est paramétrable par l'intermédiaire du panneau de configuration.

3.2.6.3.6 IDEALE Cette politique d'information a été implémentée pour permettre un étalonnage des autres. Elle simule le cas idéal où chaque site dispose d'une copie exact de l'état de charge des autres sites, sans mettre en place de mécanisme d'échange d'informations. Cette stratégie est sûrement parfaite en ce qui concerne le coût et la valeur de l'information disponible. Cependant elle

anticipe très mal la valeur de la charge sur un site dans un futur proche, ce qui entraîne des placements de processus qui ne sont pas toujours optimaux.

3.2.6.4 La politique de transfert

La politique de transfert détermine le moment où l'on déclenche une demande de localisation d'un processus à créer, ou en cours d'exécution quand le système à la possibilité de faire de la migration. La politique de transfert détermine aussi les processus qui sont examinés pour un éventuel transfert (Voir Chapitre 1). Dans notre cas la politique de transfert n'intervient que lorsqu'il y a demande de création (pas de migration), elle se cantonne dans le choix entre une politique d'équilibrage ou une politique de partage de charge.

- A chaque demande de création on recherche le meilleur site. Cette politique correspond à la mise en place d'un équilibrage de charge. On s'interroge à chaque création sur le meilleur site à choisir dans le but d'éviter une trop grosse disparité entre les indicateurs de charge.

- La recherche du meilleur site n'est faite que lorsque la charge locale dépasse un certain seuil *Seuil*. Cette politique correspond à la mise en place d'un partage de charge. On considère que tant que l'indicateur de charge locale ne dépasse pas un certain seuil, alors la création est locale. La variable *Seuil* est paramétrable par l'intermédiaire du panneau de configuration.

3.2.6.5 La politique de localisation

La politique de localisation définit le choix du site au moment de la demande d'une création que la politique de transfert a jugé bon de déplacer sur un autre site. Les politiques de localisation que nous avons implémentés se regroupent en trois classes. Une classe regroupe les choix en aveugle c'est à dire qu'il n'est pas tenu compte de la charge des sites pour faire un choix, celui-ci est fait par tirage aléatoire ou de manière cyclique. Dans une autre classe on regroupe les algorithmes qui calculent une fonction de coût faisant intervenir la charge du site, la distance qui les sépare, et la période depuis laquelle on dispose de l'information de charge. Enfin une dernière classe introduit le type de l'entité à placer pour déterminer dynamiquement le choix de l'algorithme qui va calculer la localisation.

Quelque soit la politique de localisation choisie, celle-ci ne s'applique que sur le sous-ensemble des sites susceptibles de pouvoir les accueillir. L'algorithme de construction du sous-ensemble des sites pouvant accueillir l'entité à créer est le suivant :

```
proc selection(entité, Tab_site[0..Nb_sites], j)
    j ← 0
    pour i ← 0 :→ Nb_sites faire
        si mémoire_disponible(i) ≥ Taille(entité) alors
            si code_entité(entité, i) alors
                si vérification_contraintes(entité, i) alors
                    Tab_site[j] ← i
                    j ← j + 1
                fsi
            fsi
        fsi
    fait
```

Cette procédure *selection* a pour but de garantir d'une part que le site dispose encore d'assez de mémoire pour stocker l'environnement local de l'entité. Elle permet de garantir que le code de l'entité est disponible sur le site. Ceci dans l'hypothèse d'une architecture hétérogène, ou dans le cas où le code n'a pas été chargé sur la totalité des sites. Enfin elle assure que les contraintes spécifiées par l'utilisateur dans la partie spécification de l'entité sont respectées.

- Choix de la localisation en aveugle: ALEATOIRE et CYCLIQUE

  Dans le cas d'un choix *Aléatoire*, aucune politique d'information n'est utilisée, étant donné que le choix est déterminé par tirage d'un numéro de site parmi l'ensemble des sites susceptibles d'accueillir l'entité à créer. Contrairement à [Sta85] ce choix n'est pas remis en question par le site destinataire. Cette politique va permettre de faire des comparaisons par rapport aux autres solutions qui elles, mettent en place un véritable mécanisme de placement dynamique.

  La politique de localisation *Cyclique* est une première façon de mettre en place un placement dynamique. Comme pour *Aléatoire*, le choix est fait sans prendre en compte les informations de charge.

  Remarque: *Dans ce cas seules les contraintes statiques exprimées par l'utilisateur pourront être garanties étant donné qu'il n'y a pas de politique d'information.*

- Choix du site en fonction d'un calcul du coût $f(i,j)$, avec i le numéro du site demandeur d'une création, et j un des sites destinataires possibles.

  Le choix du site est le résultat de la recherche du site j tel que:

$$f(i,j) < f(j,k),\ \forall k \in 1 \ldots Nb\_sites,\ et\ i \neq j \neq k$$

	Calcul	Création	Attente	Message	Nombre
CRO	Peu	Beaucoup	Beaucoup	Beaucoup	Peu
CGA	Peu	Jamais	Beaucoup	Beaucoup	Variable
CEM	Beaucoup	Variable	Peu	Peu	Beaucoup

TAB. 3.1 - *Caractéristiques des Cacs en terme de charge*

La fonction contient différents paramètres auxquels on applique des coefficients permettant de les inclure dans une même formule.

Ces paramètres sont :

- **l'indicateur de charge** du site k,
- **la distance** qui sépare le site demandeur i du site examiné k,
- **le temps** depuis lequel on dispose de l'information pour le site k sur le site i sur laquelle on base notre calcul,
- **la puissance** relative d'un site k par rapport aux autres.

on obtient la formule suivante :

$$f(i,j) = \alpha(tab\_IC[j] + \delta\Theta(tab\_IC[j])) + \beta d_{i,j} + \gamma\Delta(tab\_IC[j])$$

avec i le numéro du site demandeur, *tab_IC* le tableau des indicateurs de charge sur le site i, $d_{i,j}$ la distance qui sépare les sites i et j, et $\Delta(tab\_IC[j])$ la période écoulée depuis la dernière modification de la valeur de $tab\_IC[j]$. $\Theta(tab\_IC[j])$ permet d'unifier la valeur de l'indicateur de charge pour une architecture hétérogène.

En modifiant les valeurs de α, β, γ, δ on privilégie telle ou telle partie du calcul de coût. Par exemple si on prend la configuration $\alpha = 1$, $\beta = \gamma = \delta = 0$ on ne fait intervenir que l'indicateur de charge dans la détermination de la localisation. Avec la configuration $\alpha = 1$, $\beta = 5$ et $\gamma = \delta = 0$ on privilégie la distance sur l'indicateur de charge. L'intérêt de faire intervenir la date dans le calcul peut apparaître quand la politique d'information est apériodique comme dans le cas de *StratPass*. Les valeurs de α, β, γ, δ peuvent être positionnées dans le panneau de configuration (voir figure 3.14).

- Choix du site en fonction du type de l'entité.

  Les entités manipulées sont une description des Cacs du projet PVC/BOX. Si on reprend la terminologie utilisée dans la thèse de L. Courtrai [Cou92], on distingue trois types de Cacs : *CRO, CEM, CGA* qui ont chacun des particularités de comportement en terme de charge (Voir Tab. 3.1).

Comportement adopté vis à vis du type d'entité

- Les Cacs de type *CRO* qui sont les représentants d'objets sont créés dans leur majorité au lancement de l'application. Ce type de Cac génère une charge en création de Cacs et en communication. Leur localisation est déterminée de manière cyclique pour les raisons suivantes :
  1° Leur période de création (lancement de l'application) correspond à une forte demande en création, cela réduit donc le temps de réponse, de plus l'information disponible sur les indicateurs de charge des autres sites est très certainement erronée en raison du fort taux de création.
  2° La répartition cyclique des *CRO*, entraine la répartition des demandes de création, et des communications qui représentent une grande partie de leur charge.
- Les Cacs de type *CGA* qui représentent les gérants d'attributs sont créés par les *CRO*. Ce sont les Cacs qui sont créés juste après les *CRO* et avant leur utilisation par les *CEM*. Comme pour les *CRO*, leur localisation est déterminée de manière cyclique. En effet, la problématique de leur placement est équivalente à celle des *CRO* quand il sagit de répartir le coût de communication.
- Les Cacs de type *CEM* qui représentent l'exécution des méthodes sur les instances d'objets, sont créés pendant l'exécution de l'application. Ils ont une durée de vie réduite vis à vis du temps d'exécution de l'application. Ce sont eux qui justifient l'emploi des stratégies de placement dynamique. Leur placement va donc dépendre à la fois de l'état courant du système, et des informations obtenues lors des exécutions précédentes, comme la charge de calcul, le volume de communication, la charge créée.

Le type de l'entité est indiqué dans le source de l'application, dans la description de l'environnement par le champ `type`.

```
1 eratos(racine,pere,valeur)
2 {
3   taille := 1000;
4   type := CEM;  -- type de l'entite
5
6   var donnee, rqt, next, fini, q, r,cal;
7
8   begin
```

3.2.6.6 Présentation de la méthode du GRADIENT

Notre solution diffère au niveau de la détermination de la localisation par rapport à la présentation faite au chapitre 1. En effet dans [LK87], le mécanisme de migration est utilisé pour répartir la charge entre sites. Dans notre cas, cette répartition n'a lieu qu'au moment de la création d'une entité. Si le site demandeur est dans l'état *Libre* ou *Actif*, la création est faite localement, si le site est chargé, l'entité qui fait la demande de création envoie un message à la procédure *lb(i)* (voir fig. 3.18) du site et se bloque en attente de la réponse. La recherche du meilleur site est faite de proche en proche, par l'intermédiaire du voisin qui a la valeur *pp(i)* minimale. La recherche s'arrête quand on a dépassé une certaine distance (*Rech_Max*) par rapport au site demandeur, ou quand on a trouvé un site *Libre*.

3.2.7 Les résultats de simulation

La simulation avec gestion d'une horloge logique a l'avantage de faciliter l'enregistrement des informations pour faire une analyse détaillée de l'exécution. Le travail des *espions* chargés de recueillir les informations n'étant pas comptabilisé.

Très souvent l'analyse des performances d'une exécution se fait en *post-mortem* [HE91, KTP93] pour éviter de trop perturber l'exécution de l'application. Avec notre plateforme, il est possible de visualiser dynamiquement l'évolution de certaines données de l'exécution (Voir Fig. 3.20). Le développement d'outils graphiques va faciliter le travail d'analyse de l'observateur sur le volume des informations recueillies.

Comme le montre la figure 3.19, deux types d'informations sont obtenus à la suite d'une exécution. Les informations par site d'une part qui donnent un compte rendu de son activité en répartissant le temps logique consommé par l'exécution des entités, le traitement de la politique d'échange d'information, et le temps où le site a attendu du travail. Les informations par entité et par relation entre les entités d'autre part.

3.2.7.1 Les informations sur un site

Ces informations vont permettre de comparer les exécutions d'une application en modifiant les paramètres de la stratégie de placement, et notamment le temps en nombre d'unités logiques pour l'exécution de l'application (voir Fig 3.21).

- Le temps en nombre d'unités logiques pour l'exécution de l'application;
- Le temps où le site était en attente de travail (aucune entité prête);

```
proc lb(i)
    tq (1) faire
        attente_message_lb(rqt, e, n, d)
        Etat ← Valeur_Etat()
        choix rqt faire
            cas Demande :
                min ← Rech_Max
                pour i = 0 :→ Nb_voisins faire
                    si pp(i) < min alors Dest ← i fsi
                fait
                si (min < Rech_Max) ∧ (Rech_Max > 1) ∧ (min ≠ 0) alors
                    envoi_message lb(Dest) (Question, e, n, Rech_Max − 1)
                sinon
                    si min ≠ 0 alors
                        envoi_message e (n)          - pas de solution proche
                    sinon
                        envoi_message e (Dest)       - Le voisin est Libre
                    fsi
                fsi
            cas Question :
                si ETAT = Libre alors
                    envoi_message lb(n) (Réponse, e, i, 0)
                sinon
                    min ← d
                    pour i = 0 :→ Nb_voisins faire
                        si pp(i) < min alors Dest ← i fsi
                    fait
                    si (min < d) ∧ (d > 1) ∧ (min ≠ 0) alors
                        envoi_message lb(Dest) (Question, e, n, d − 1)
                    sinon
                        si min ≠ 0 alors
                                                     - pas de solution proche
                            envoi_message lb(n) (Réponse, e, n, 0)
                        sinon
                                                     - Le voisin est Libre
                            envoi_message lb(n) (Réponse, e, Dest, 0)
                        fsi
                    fsi
                fsi
            cas Réponse :
                envoi_message e (n)                  - envoi de la solution
        fchoix
    fait
```

FIG. 3.18 - *Procédure de recherche du meilleur site dans la stratégie du Gradient*

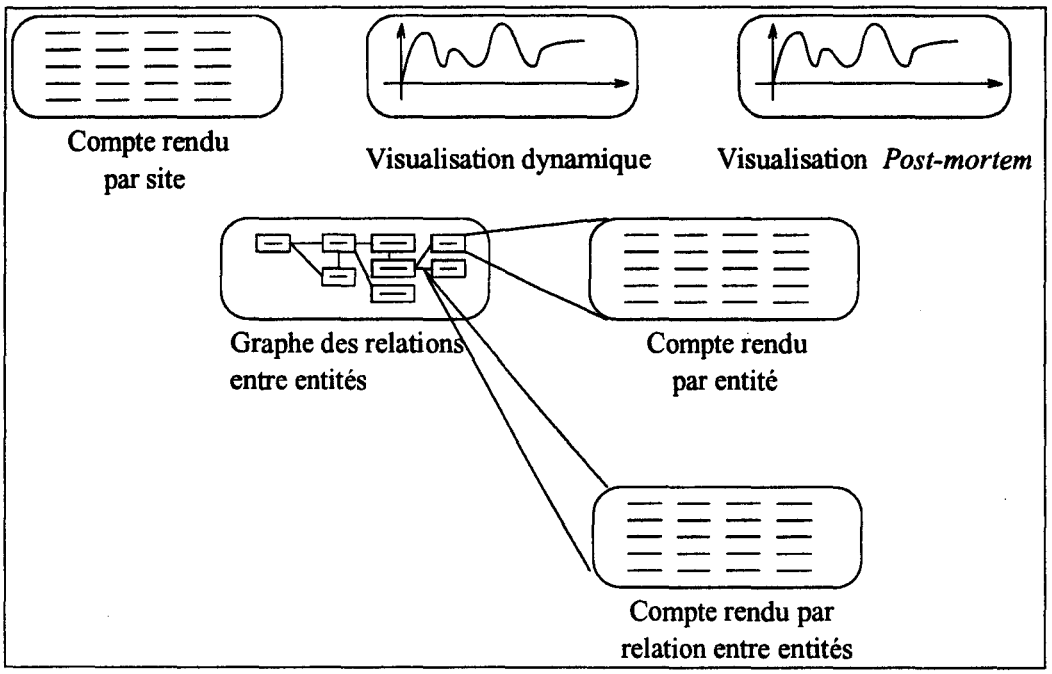


FIG. 3.19 - *informations recueillies suite à la simulation*

- Le nombre de demandes de création;
- Le nombre de créations;
- Une estimation du coût de la politique d'échange d'informations;
  - le nombre de messages reçus
  - le nombre de messages envoyés
  - le coût de communication produit
  - le coût d'exécution
- le coût de communication produit par les entités;
- le coût d'exécution des entités;
- le maximum de mémoire occupée

3.2.7.2 Les informations sur une entité et ses relations

Les informations sur les entités et leurs relations permettent de définir un graphe de communication, un graphe de dépendance, et de fournir une estimation du coût d'exécution de chaque entité (voir les Fig. 3.22 & 3.23).

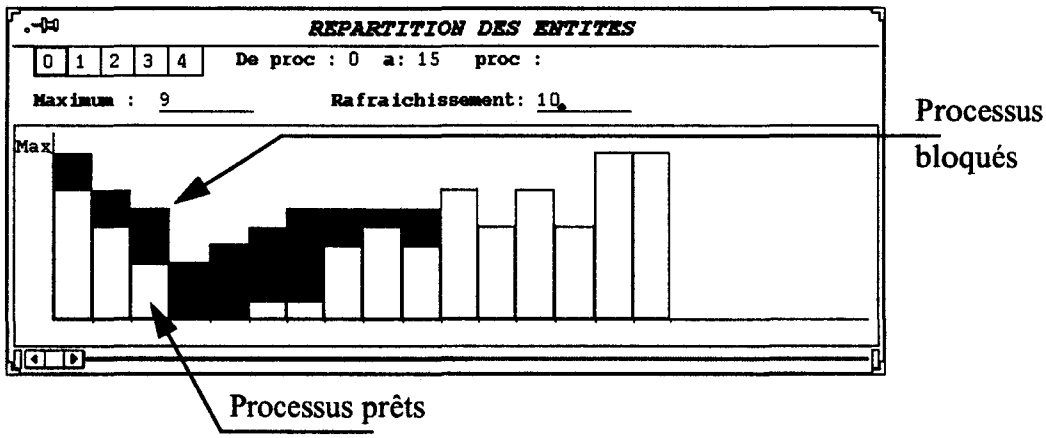


FIG. 3.20 - *Visualisation de l'évolution dynamique de la charge sur les sites*

1° Les informations par entité;

- le nombre d'exemplaires créés;
- le nombre d'exemplaires existant simultanément;
- la répartition des exemplaires créés et existant simultanément sur les sites;
- le coût de calcul;
- la durée de vie;
- le coût de communication produit

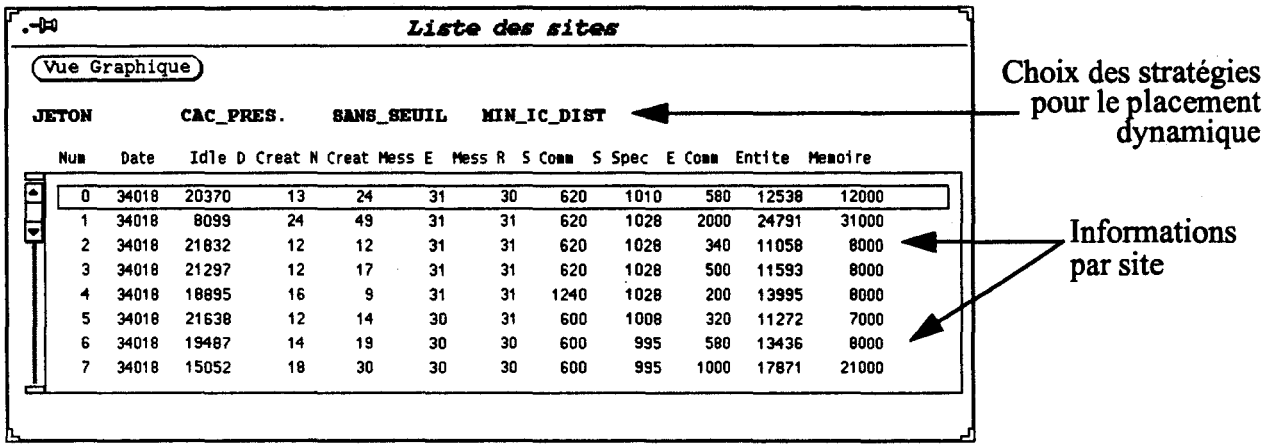
2° Les informations sur les relations entre une entité A et une entité B

- le nombre de créations d'entité B par A;
- le nombre de messages reçus par l'entité A de l'entité B;
- le nombre de messages émis par l'entité B pour l'entité A;
- le coût de la communication produit par les messages émis

3.3 Conclusion

Notre objectif était de fournir une plateforme d'évaluation, permettant d'étudier le comportement de l'exécution d'une application en fonction de son découpage en *Composants Actifs Communicants* et en fonction de différentes stratégies de placement dynamique.

Pour cela nous avons proposé une modélisation permettant de simuler le modèle d'exécution correspondant au projet PVC/BOX. Cette modélisation devait



Num	Date	Idle D	Creat N	Creat Mess	E Mess R	S Comm	S Spec	E Comm	Entite	Memoire
0	34018	20370	13	24	31	30	620	1010	580	12538	12000
1	34018	8099	24	49	31	31	620	1028	2000	24791	31000
2	34018	21832	12	12	31	31	620	1028	340	11058	8000
3	34018	21297	12	17	31	31	620	1028	500	11593	8000
4	34018	18895	16	9	31	31	1240	1028	200	13995	8000
5	34018	21638	12	14	30	31	600	1008	320	11272	7000
6	34018	19487	14	19	30	30	600	995	580	13436	8000
7	34018	15052	18	30	30	30	600	995	1000	17871	21000

FIG. 3.21 - *Présentation des informations sur un site*

pouvoir être évaluée sur une gamme d'architectures englobant l'ensemble des machines de type multicomputer (Voir chapitre 1). Nous avons implémenté un langage de description synthétique de Cacs permettant de tracer les événements importants d'une exécution en terme de charge pour ce modèle d'exécution, afin de faciliter l'analyse de l'exécution d'une application.

Un simulateur a été implémenté, comportant un certain nombre de stratégies de placement dynamique pouvant satisfaire au problème du placement des Cacs. Toutes ces stratégies s'articulent autour de quatre composantes largement paramétrables i) l'évaluation de la charge d'un site, ii) la politique d'échange d'information, iii) la politique de transfert, iv) la politique de localisation. De plus elles mettent en place un système distribué de décision basé sur une duplication plus ou moins globale de l'information de charge du système.

Reste à conduire une campagne d'analyses des performances qui doivent nous permettre d'apporter à l'utilisateur une méthode et un outil de détermination des paramètres de découpage de l'application et des paramètres de la stratégie de placement dynamique, cela en fonction de l'architecture de la machine parallèle et des caractéristiques de l'application. C'est ce que nous allons aborder dans le chapitre suivant.

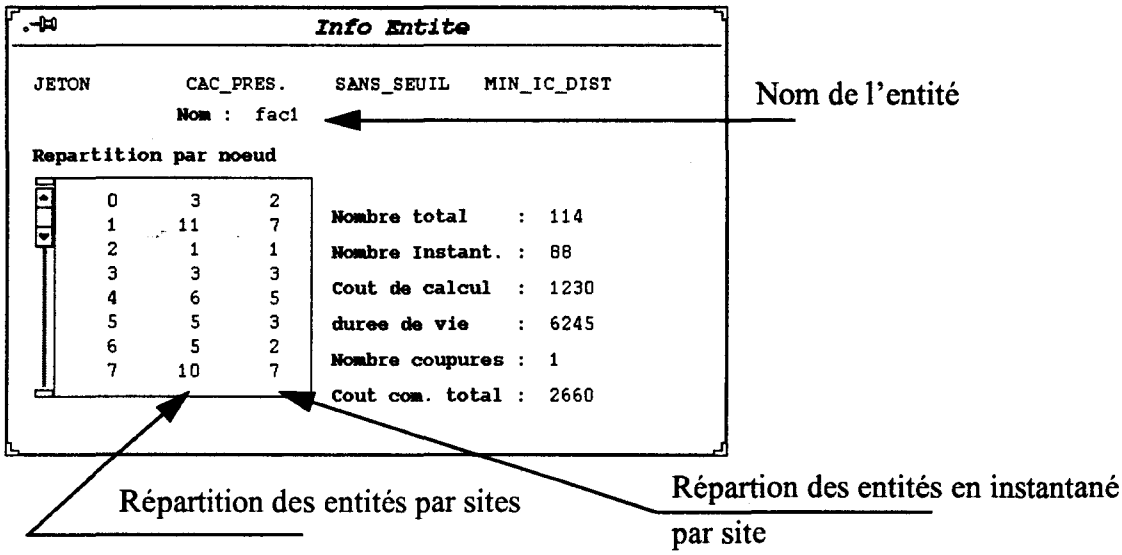


FIG. 3.22 - *Présentation des informations sur une entité*

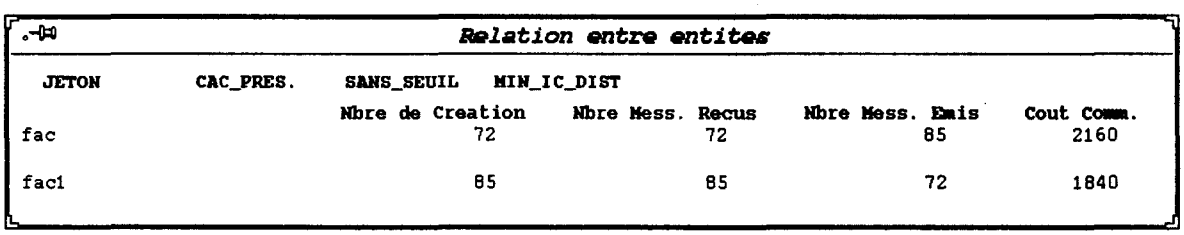


FIG. 3.23 - *Présentation des informations sur les relations entre entités*

Chapitre 4

Etude du placement dynamique d'applications caractéristiques avec la plate-forme d'évaluation

4.1 Introduction

Dans les chapitres précédents, après avoir présenté le problème général du placement, précisé le contexte plus particulier de la recherche d'un placement pour un ensemble de processus communicants, nous avons proposé une plate-forme de simulation qui nous permet d'étudier le comportement des stratégies de placement les plus adaptées à notre projet (PVC) et son modèle d'exécution, cela en fonction d'architectures de machine parallèle et d'applications.

Dans ce chapitre nous allons proposer l'analyse du comportement de certaines applications caractéristiques et simples qui montreront la démarche utilisée pour déterminer les paramètres d'une stratégie de placement dynamique adaptées. Pour cela nous allons commencer par déterminer les paramètres globaux qui vont permettre de simuler le comportement des architectures cibles des machines parallèles envisagées.

4.2 Description des architectures

Le module de spécification d'une architecture dans notre plate-forme nous permet de définir les coûts relatifs de chacune des instructions tracées, ainsi que le coût de communication inter-sites. Pour fixer ces paramètres globaux nous avons utilisé des résultats de mesures effectuées entre autres sur la machine parallèle du LIFL (un MultiCluster II de chez Parsytec) et un réseau de stations de travail (Sun sparc)

4.2.1 Détermination des paramètres globaux de l'architecture

Une architecture réelle est modélisée dans notre plate-forme de simulation par 3 niveaux de description (voir le paragraphe 3.2.5) : la description d'un site, la description de la topologie du réseau d'interconnexion, et enfin des paramètres globaux de l'architecture.

Nous avons mis en place une série de mesures sur l'architecture réelle, de façon à déterminer les constantes qui seront utilisées pour fixer les paramètres globaux repris dans notre plate-forme de simulation. Les mesures nous ont permis de préciser les caractéristiques qui nous intéressent :

- Le coût d'une mise en forme d'un message jusqu'à sa transmission au système de communication : EM;
- Le coût de la lecture d'un message présent dans la boîte aux lettres d'un processus : RM;
- Le coût de communication entre deux noeuds voisins: CC_u;

  Pour déterminer une valeur moyenne de ces différents coûts, nous avons fait varier la taille des messages échangés dans un intervalle représentatif d'une utilisation normale de l'architecture (16 octets — 4 Koctets). Les valeurs obtenues font apparaître un écart type très faible, ce qui est en accord avec le choix de faire une approximation de leur valeur par une constante.
- Le coût d'une demande de création d'un processus;

  Comme nous l'avons vu dans le chapitre précédent, la création nécessite la mise en place d'un mécanisme de question-réponse synchrone entre le noeud demandeur de la création et le noeud élu pour recevoir le processus créé. Soit $CCP_{i,j}$ le coût d'une demande de création d'un processus sur le site i ayant choisi j comme site pour créer le processus. Le coût se décompose en trois traitements :

  1. La détermination du site j : $CRMN$,

2. Un échange de messages pour interroger le site choisi j et obtenir la référence du processus créé,
3. Le coût de traitement de la création du processus est à "*facturer*" au site j, soit $CTCP$ ce coût. $CTCP$ représente le coût de réservation de l'espace mémoire pour le processus.

Le coût d'une demande de création d'un processus englobe les coûts (1) et (3) ($CRMN+CTCP$) avec un rapport de 10% pour (1) et de 90% pour (3). L'échange de messages est pris en compte par le coût de communication.

- Le coût de calcul.

  Nous avons choisi comme valeur étalon pour indiquer un coût de calcul une boucle d'affectation du résultat de la multiplication d'une variable par une constante réelle, soit CE cette valeur étalon.

```
max = 100000;
for (i = 0 ; i < max; i++)
  a = b * 3.14;
```

Les résultats présentés dans la suite, n'ont pas valeur d'une mesure de performance, l'utilisation des différents niveaux de systèmes permettant difficilement d'en faire une évaluation exacte, leur étude demanderait un travail bien plus précis. Ce qui nous intéresse ici, c'est de pouvoir faire une comparaison entre les différentes grandeurs qui sont utilisées par la plate-forme de simulation.

4.2.2 Résultats des mesures

Le tableau 4.1 synthétise les résultats de mesures faites à la fois sur le MultiCluster II et le réseau de stations de travail. Les lignes PGA correspondent aux valeurs des Paramètres Globaux de l'Architecture qui seront utilisées comme entrées de la plate-forme de simulation. Elles correspondent à une normalisation des mesures obtenues. Cette normalisation consiste à ramener à une valeur 100 le coût de création d'un processus et à appliquer le même rapport aux autres valeurs.

Mesures sur le MultiCluster II.

Cette machine à base de transputers (32 T800) est totalement reconfigurable, ce qui permet d'envisager toutes sortes d'architectures ayant un degré inférieur ou égal à quatre.

Machine	EM	RM	CC_u	CE	$CRMN + CTCP$
MC-II	155	96	233	48	679
PGA MC-II	26	16	39	8	100
Rés. Stations	15	26	10	3	302
PGA Rés. Stations	5	7	4	1	100

TAB. 4.1 - *Résultats des mesures effectuées*

Le run time de notre modèle d'exécution PVC ayant été implémenté, nous avons effectué ces mesures au-dessus de l'implémentation du run-time PVC, afin de déterminer une combinaison de paramètres vraisemblable pour notre architecture simulée.

Les tests ont été réalisés en écrivant une application utilisant une implémentation de notre run-time. Par conséquent, les valeurs obtenues sont une évaluation du run time PVC et non une évaluation du MultiCluster II. Pour limiter les erreurs nous avons réalisé les mesures en faisant une boucle d'itérations d'une séquence réduite d'instructions.

Mesures sur un réseau de stations de travail.

Les mesures ont été faites sur un réseau de stations de travail, dans les mêmes conditions que précédemment.

Remarques sur les résultats obtenus.

— Les deux architectures de machines ont un rapport coût de communication, coût de calcul qui est équivalent (5:1) pour la MC2, et (4:1) pour le réseau de stations. Si on considère le coût d'un envoi de message vers un site voisin, il est moins coûteux sur la MC2 (10:1) que pour un réseau de stations (16:1).

— On constate que le coût de création est relativement plus important pour le réseau de stations de travail (100:1) que pour la MC2(12.5:1). Ceci est dû à l'utilisation de la librairie *lwp* du système d'exploitation SunOS [Sun88] qui pénalise la création de nouveaux *threads* par un coût prohibitif de l'algorithme d'allocation mémoire.

La notion de distance disparait quand on travaille sur un réseau de stations de travail. Pour deux machines i et j, la distance vaut $D_{i,j} = 1$ si $i \neq j$ et 0 sinon.

Simulation d'une architecture virtuelle

La plate-forme de simulation permet de simuler toute une gamme de machines ayant des caractéristiques globales différentes. Ceci est possible en faisant varier

EM	RM	CC_u	CE	$CRMN + CTCP$	Observations
1	1	100	1	10	Architecture avec un coût de communication important
10	10	10	10	20	Architecture avec des coûts de communication et de calcul équivalents
50	50	100	1	5	Architecture avec un système de communication coûteux
50	50	100	1	5	Architecture avec un système de communication coûteux
50	50	10	100	10	Architecture avec un système de communication plus performant que la CPU des sites

TAB. 4.2 - *Exemples d'architectures virtuelles*

les valeurs relatives de chacun des paramètres globaux. Ce qui permet d'étudier l'évolution des temps de simulation pour une application et une stratégie de placement données. Le tableau 4.2 donne quelques exemples d'architectures virtuelles.

4.3 Outils utilisés

4.3.1 Remarques préliminaires

Dans l'ensemble des stratégies décrites dans le chapitre 3, certaines ne sont pas reprises dans les exemples qui vont illustrer ce chapitre. Dans les politiques d'information nous avons par exemple implanté les stratégies *Gradient* et *STRAT-PASS*, en fait il s'avère que ces stratégies ne sont pas adaptées d'une part à l'absence de migration de tâche, et d'autre part au modèle d'exécution utilisé dans notre projet. Dans tous les cas testés, ces deux stratégies ont donné des résultats très mauvais sans contestation possible. Dans notre modèle, il y a deux étapes pour faire la régulation de la charge, dans la première on recherche un site, puis dans la seconde on envoie une demande de création vers le site choisi. L'intérêt de la méthode du *Gradient* est de propager de proche en proche les indicateurs de charge, mais aussi les processus transférés, ce qui limite le nombre de messages échangés par la politique d'information. Une adaptation de la méthode du *Gra-*

dient à notre modèle d'exécution, conduit à propager la requête de demande de création vers le meilleur site pour la prendre en charge, ce qui allonge le temps de traitement d'une demande de création. Les gains obtenus sur la politique d'information sont perdus lors de son utilisation par la politique de localisation. Comme il est rappelé dans le paragraphe 3.2.6.3.5, les stratégies passives sont plutôt utilisées avec un mécanisme de migration. Nous espérions que l'absence de ce mécanisme dans notre modèle de répartition serait compensé par le faible coût et le nombre élevé des entités manipulées. Cependant l'utilisation d'un *Seuil* pour indiquer la valeur au-dessous de laquelle le site doit demander du travail a introduit un autre phénomène qui fait que plutôt que répartir équitablement la charge simultanément sur l'ensemble des sites, celle-ci va se déplacer par vague d'un site vers un autre.

Dans les politiques de calcul de l'indicateur de charge, la possibilité d'utiliser une valeur moyenne reprenant un ensemble de valeurs antérieures n'a pas été exploitée, des variations de charge très rapides étant constatées, mais sans que l'on puisse constater de phénomène de *pic* marginal qui aurait été gommé par le calcul d'une moyenne.

4.3.2 Les instruments de comparaisons

Nous allons utiliser un certain nombre d'indicateurs qui nous permettrons de comparer les résultats de simulations et d'en faciliter leur analyse. L'ensemble de ces indicateurs forme un échantillon des résultats obtenus suite à une exécution (Voir le paragraphe 3.2.7).

Le temps de simulation

Une première méthode pour comparer les résultats de simulation est de regarder le temps de simulation obtenu en fonction d'un paramètre qui varie tous les autres étant égaux par ailleurs. Cela nous indique la configuration la plus prometteuse des paramètres de la stratégie de répartition pour une application et une architecture données. Le temps de simulation correspond au temps local le plus grand sur l'ensemble des noeuds.

Le temps moyen d'inactivité des noeuds

Cette valeur permet de savoir si la répartition a permis de réduire les temps d'inactivité pendant l'exécution d'une application. Certaines périodes d'inactivité sont dûes aux attentes de messages et donc aux mauvais recouvrements des temps de communication par les temps de calcul, d'autres sont dûes à la mauvaise

répartition des calculs. L'inactivité moyenne est calculée en faisant la moyenne des inactivités sur l'ensemble des noeuds.

La répartition du coût de calcul: variance cumulée

La variance cumulée correspond à la somme des différences entre la charge CPU consommée par un site pour l'exécution de l'application ($Tcpu_i$) et le temps CPU qu'il aurait dû consommer si la charge avait été parfaitement équilibrée entre tous les sites ($\frac{\sum_{i=1}^{Nb\_sites} Tcpu_i}{Nb\_sites}$).

$$V = \sum_{i=1}^{Nb\_sites} \| Tcpu_i - \frac{\sum_{j=1}^{Nb\_sites} Tcpu_j}{Nb\_sites} \|$$

Plus cette valeur sera importante, moins bon aura été l'équilibrage de la charge.

Le coût du traitement de la répartition dynamique

Ce coût nous indique le sur-coût de calcul et de communication que provoque la mise en place d'une répartition dynamique. Elle se représente par deux données : 1) le coût de calcul, c'est à dire le traitement des messages spécifiques pour la mise en place de la politique d'information; 2) le coût de communication, ou encore l'utilisation du réseau de communication. Plus ces valeurs seront importantes et plus l'exécution de l'application en sera perturbée.

Le coût de communication

Il correspond à l'utilisation du réseau de communication par les processus de l'application. Si le recouvrement se fait bien entre calcul et communication, alors le coût de communication n'a pas d'incidence sur le temps d'exécution, dans le cas contraire, toute augmentation de ce coût augmente le coût d'exécution de l'application.

4.4 Présentation de quelques exemples d'applications

Dans la suite de ce chapitre, nous allons présenter un ensemble d'exemples qui vont nous permettre de mettre en lumière les caractéristiques importantes qu'il faut connaitre pour bien adapter la stratégie de placement d'une application. Ces

exemples sont volontairement simples pour permettre de bien en faire ressortir les résultats.

4.4.1 L'application factorielle

L'application factorielle n a déjà été présentée dans le chapitre 2 (cf 2.3.2.4), ses caractéristiques sont celles d'une application qui développe une structure dynamique régulière de processus qui font peu de calcul et peu de communication. L'application est composée de deux entités, la première (*root*) lance le calcul, la deuxième (*fac*) fait le calcul, en lançant récursivement la même entité. Ce programme construit un arbre binaire équilibré de processus. Chaque processus exécute le même code.

4.4.1.1 Influence de l'architecture sur l'évolution du temps de simulation

Pour l'architecture nous avons choisi une topologie de réseau en forme de tore avec 9, 16, 20, 25, 30 sites. Pour la machine avec 64 sites la topologie du réseau est une grille de dimension deux. Les paramètres globaux de l'architecture sont ceux donnés dans le paragraphe 4.2.2, et concernant le *MultiCluster-II.*

La figure 4.1 montre que le temps de simulation décroit en même temps que le nombre de sites augmente (on évite d'utiliser le terme de *speed-up* lorsqu'il sagit de simulation). La répartition dynamique utilisée correspond au cas *Idéal* où l'on dispose d'une information de charge parfaite. L'application factorielle entraîne la création d'un nombre important de processus : elle s'adapte bien à une architecture disposant de nombreux noeuds, le seuil étant le nombre de feuilles de l'arbre des processus.

4.4.1.2 Influence de l'indicateur de charge

Les simulations ont été faites en utilisant une architecture homogène. Nous avons retenu une machine composée de seize sites utilisant un réseau d'interconnexion représentant un tore.

La figure 4.2 présente les résultats de simulation en fonction de l'indicateur de charge, avec *Ent. Prés.* qui représente le nombre d'entités présentes, *Ent. Act.* le nombre d'entités actives. *Moy. Ent. Prés.* et *Moy. Ent. Act.* utisent les mêmes informations en appliquant un calcul de moyenne sur une période de temps. La politique d'information est *Idéal* et la politique de localisation est *Min_IC_D* (Voir paragraphe suivant). Le temps minimal de simulation est obtenu avec l'indicateur de charge *Ent. Prés.*, qui minimise aussi le temps de communication inter-sites,

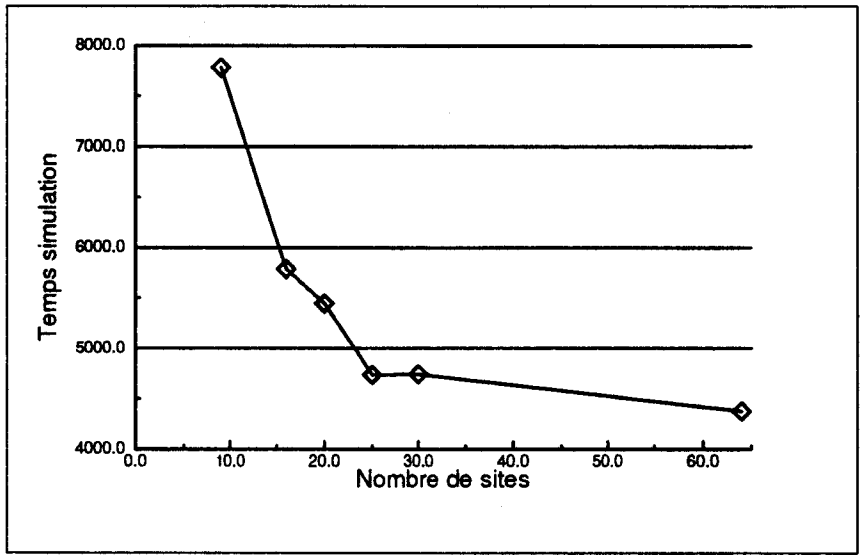


FIG. 4.1 - *Résultats pour factorielle en fonction de l'architecture*

le temps moyen d'inactivité, et donne un bon équilibrage de la charge sur les sites. L'indicateur de charge *Ent. Act.* équilibre moins bien la charge ce qui provoque une augmentation du temps moyen d'inactivité des sites. Dans la suite, les simulations utilisent *Ent. Prés.* comme indicateur de charge.

4.4.1.3 Influence de la politique d'information et de la politique de localisation

Le but du placement dynamique est d'établir une répartition équitable de la charge sur l'ensemble des sites, l'algorithme de choix du placement d'une entité est donc déclenché à chaque demande de création.

Le choix du site se fait, soit en aveugle pour *Aléatoire* ou *Cyclique*, soit en utilisant *Min_IC_D* qui utilise la formule présentée dans le chapitre 3 (cf 3.2.6.5) avec $\alpha = CC_{i,j}$ avec $Di, j = 1$, $\beta = 1$, $\gamma = 0$, $\delta = 1$.

Le choix de la politique d'information est le plus largement étudié car il constitue le maillon le plus délicat d'un placement dynamique. Nous présentons d'abord le cas idéal où l'information de charge est disponible directement sans mettre en place de politique d'information qui est noté *Min_IC_D* dans la figure. Ensuite sont présentées les politiques d'informations *Jeton, Message, Message Plus, Threshold, Broadcast* qui utilisent toutes, la politique de localisation *Min_IC_D.*

Dans la figure 4.3 sont données différentes courbes permettant d'analyser le déroulement de l'exécution de l'application en fonction de l'échantillon du paramétrage utilisé des stratégies de placement dynamique.

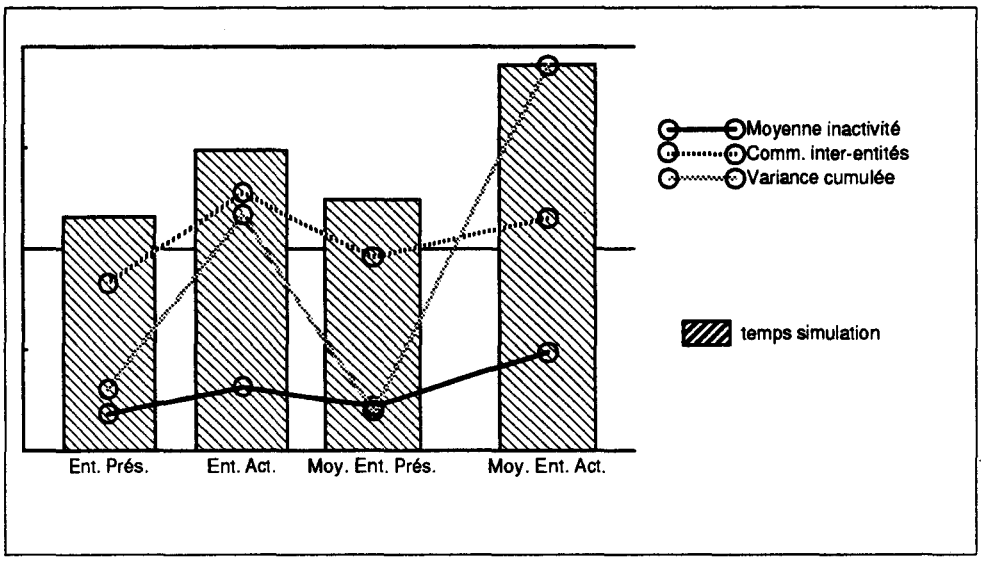


FIG. 4.2 - *Résultats de simulation en fonction de l'indicateur de charge*

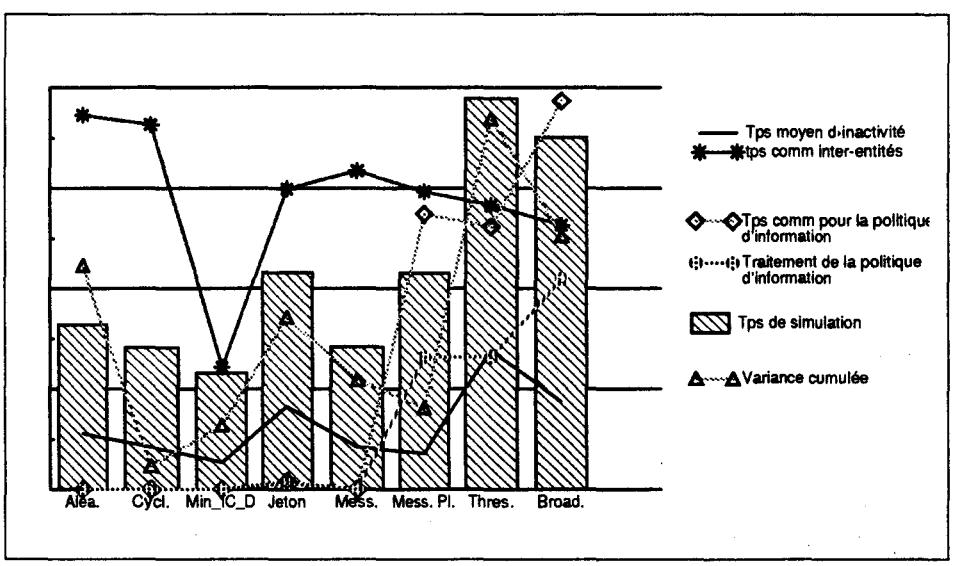


FIG. 4.3 - *Résultats pour factorielle en fonction de la stratégie de placement*

On trouve sous forme d'histogramme le temps de simulation (en unité de temps logique) de l'application factorielle. On constate que le temps de simulation le plus court est obtenu par la stratégie *Min_IC_D* ce qui tend à prouver que le placement dynamique conduit à une meilleure utilisation des ressources. Pour les stratégies utilisables dans un système réel, la politique d'information *Message* conduit à une bonne utilisation des ressources juste devant *Aléatoire* et *Cyclique*, viennent ensuite les stratégies de placement utilisant des politiques d'informations qui pénalisent l'exécution de l'application du fait du coût des échanges de messages nécessaires à leur mise en place.

La figure 4.3 présente aussi des courbes donnant des indications sur le coût de la mise en place de la politique d'information, notamment le temps de communication et le temps CPU consommé. On remarque que ces temps sont les plus importants pour la politique d'information *Broadcast*, cela est dû au fait que l'on considère que l'implémentation de l'envoi de message vers plusieurs destinataires se fait par l'envoi successif de messages, donc très coûteux; c'est cependant le cas de la machine parallèle disponible au LIFL (MultiCluster II). Viennent ensuite *Message Plus, Threshold, Jeton* étant la moins coûteuse. Le coût de la politique d'information de *Message* est nul car elle n'ajoute pas de messages d'information au trafic des communications, et on considère le sur-coût de traitement des messages pour prendre en compte l'information de charge comme négligeable.

Trois autres courbes présentent comment ont été utilisées les ressources (CPU essentiellement) au cours de l'exécution.

Une première courbe nous indique le temps moyen d'inactivité des sites au cours de l'exécution; plus ce temps est faible et plus l'utilisation des ressources aura été judicieuse. On constate par exemple que *Broadcast, Threshold* qui sont coûteuses ne conduisent pas à une utilisation optimale des ressources, et conduisent à des temps d'inactivité supérieurs à ceux produits par *Aléatoire* ou *Cyclique*.

Pour la courbe représentant la variance cumulée la stratégie *Cyclique* donne le meilleur résultat, on pouvait s'y attendre étant données les caractéristiques de l'application à placer. Les mauvais résultats de *Broadcast* et *Threshold* peuvent s'expliquer là encore par le temps CPU important utilisé par les sites pour mettre en place la politique d'information, ce qui induit un délai plus important pour acheminer l'information, et par conséquent une perte de la pertinence de l'information.

La courbe du coût de communication entre entités est plus spécifique à l'application. Les stratégies comme *Aléatoire* ou *Cyclique*, qui placent sans tenir compte de la charge de l'application, donnent les temps les plus élevés.

En conclusion, le placement de factorielle n telle qu'elle est implémentée ne demande pas une stratégie de placement dynamique complexe, qui nécessite la mise en place d'un mécanisme d'échanges d'information coûteux et qui de plus

ne donne pas un meilleur placement des entités. Les stratégies de placement en aveugle sont meilleures pour la bonne raison que l'application simulée engendre une structure régulière (un arbre binaire équilibré) constituée de processus tous identiques, ce qui fonctionne bien avec des stratégies de placement régulières et systématiques comme *Aléatoire* et *Cyclique.*

Ces résultats sont comparables à ceux obtenus par les mesures faites après exécutions sur la machine parallèle *MultiCluster II.* Nous allons apporter, comme dans le chapitre 2 (cf 2.3.2.4), les modifications qui introduisent des phénomènes d'irrégularité dans l'arbre binaire de processus engendré.

Analyse des résultats de l'application factorielle n modifiée

Nous avons modifié l'application factorielle en deux étapes. Dans la première étape nous avons fait varier le coût de calcul comme présenté dans le tableau 4.3. Dans une deuxième étape nous avons augmenté les coûts comme présenté dans le tableau 4.4.

entité	fac	fac1	fac2
coût	10	100	1000

TAB. 4.3 - *Coût de calcul pour la première étape*

entité	fac	fac1	fac2
coût	100	1000	5000

TAB. 4.4 - *Coût de calcul pour la deuxième étape*

Nous avons fait deux séries de simulation pour chaque étape. Dans la première nous avons continué à utiliser le nombre d'entités présentes comme indicateur de charge. Dans la deuxième nous avons utilisé le coût relatif des entités comme indicateur de charge. La valeur de ce coût relatif est fournie dans le source de l'application simulée par le programmeur; elle est obtenue grâce à l'analyse d'une première simulation.

Les résultats se présentent sous la forme de quatre graphiques en forme d'histogrammes (Fig. 4.4 & Fig 4.5). Chacun des graphiques permet d'étudier le comportement de l'exécution de l'application en fonction du paramétrage de l'algorithme de placement dynamique.

Que ce soit dans la Figure 4.4 ou dans la Figure 4.5 on constate que les stratégies qui font du placement en aveugle (*Aléatoire, Cyclique*) donnent des temps d'exécution plus longs que les autres stratégies. Le comportement de l'application n'étant plus régulier, le placement conduit à une répartition mal équilibrée de la charge, ce qui induit un temps d'exécution plus long.

Dans la première étape (Fig. 4.4), on constate que le choix du nombre d'entités présentes (sans coût) comme indicateur de charge donne un meilleur résultat que celui utilisant le coût différencié des entités(avec coût). Les temps d'inactivité des

sites sont en augmentation, de même que les temps de communication entre les entités. Par contre la variance cumulée est moins élevée, ce qui veut dire que la charge de calcul est généralement mieux répartie.

Ce résultat s'explique par le fait que l'intégration du coût différencié des entités dans l'indicateur de charge favorise une meilleure répartition équilibrée de la charge de calcul, mais cela au détriment des temps de communication. Si comme dans notre cas les entités attendent des messages, alors les sites sont inactifs.

Dans la deuxième étape (Fig. 4.5), les coûts de calcul sont supérieurs. On a toujours une meilleure répartition de cette charge de calcul, cependant si comme dans la première étape les coûts de communication augmentent, ce coût est recouvert par une activité accrue des sites pour traiter la charge calcul. C'est pourquoi dans ce cas on obtient des résultats de simulation globalement meilleurs.

4.4.1.4 Conclusion sur le placement de l'application factorielle

L'application factorielle est caractéristique à plusieurs égards. C'est une application qui engendre un taux de création d'entité au cours de l'exécution qui est important, et qui se maintient pendant une bonne partie de l'exécution. La structure qu'elle développe correspond à un arbre binaire bien équilibré. Les coûts de chaque instance d'entité sont égaux, seule la durée de vie est variable suivant que l'instance se trouve plutôt dans le haut de l'arbre ou dans le bas.

Le problème du découpage de l'application ne se pose pas, car il n'y a qu'un seul type d'entité. Le placement de l'application factorielle ne nécessite pas la mise en place d'une stratégie complexe de placement. La charge qu'elle engendre est régulière, il suffit donc d'un algorithme de placement en aveugle cyclique ou aléatoire pour obtenir une bonne utilisation des ressources. Cependant dès qu'une modification intervient dans la répartition des charges engendrées par chaque instance d'entité, on s'aperçoit alors que les résultats du placement en aveugle deviennent catastrophiques (Fig. 4.4 & Fig 4.5).

4.4.2 Le problème des huit reines

Le problème consiste à rechercher les solutions du placement de N reines sur un échiquier de NxN cases, sachant que deux reines ne peuvent se trouver sur la même ligne, la même colonne, ou la même diagonale.

Nous avons implémenté un algorithme parallèle de recherche de l'ensemble des solutions (92 solutions pour un échiquier 8x8) tiré de [Ath87]. Comme il ne peut y avoir qu'une seule reine par ligne, ou par colonne, l'algorithme peut rechercher en parallèle par ligne ou par colonne. Si on choisit de travailler par colonne, on lance en parallèle sur l'ensemble des lignes une recherche des solutions en parcourant

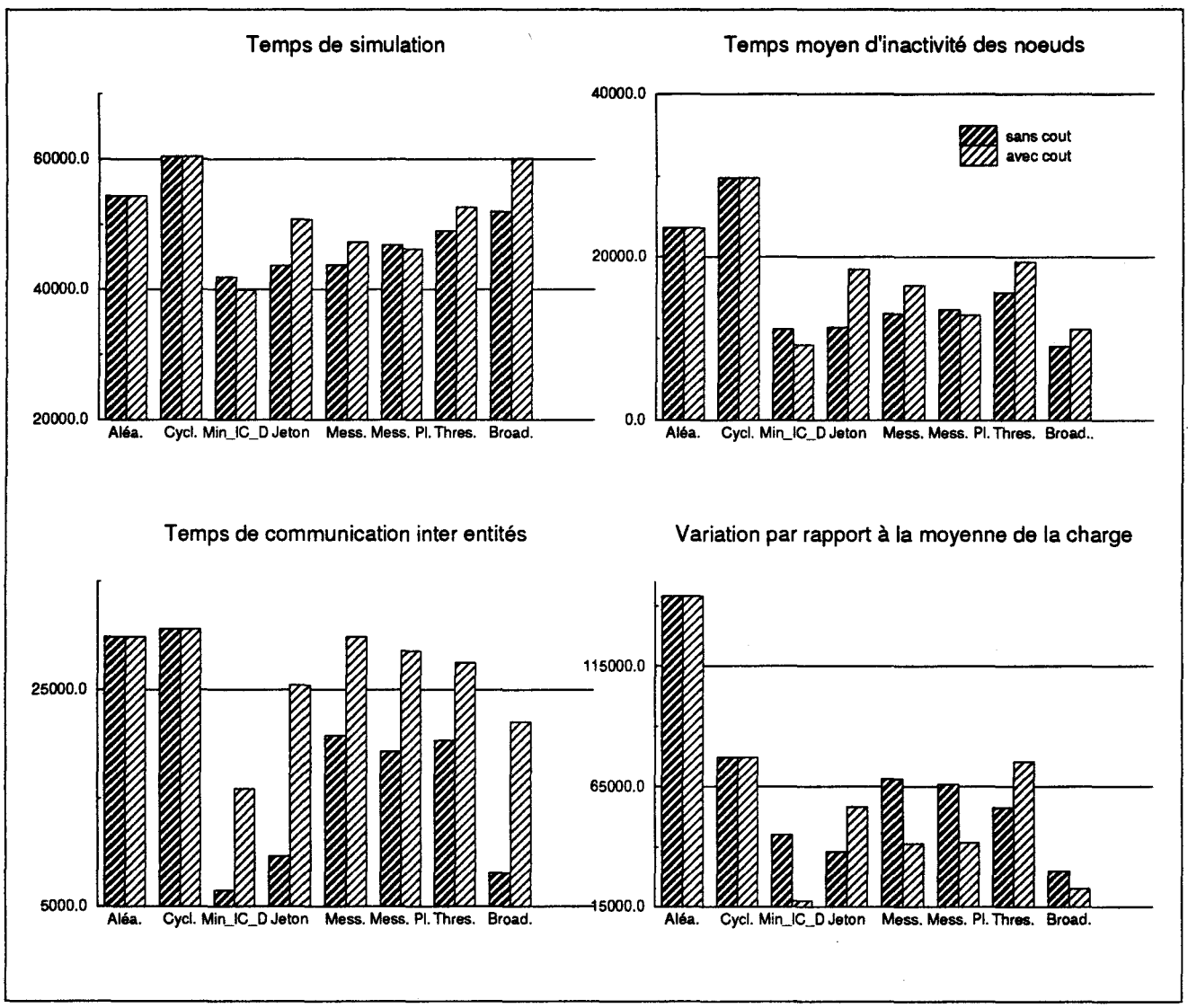


FIG. 4.4 - *Analyse des résultats de simulation de factorielle n modifiée (1)*

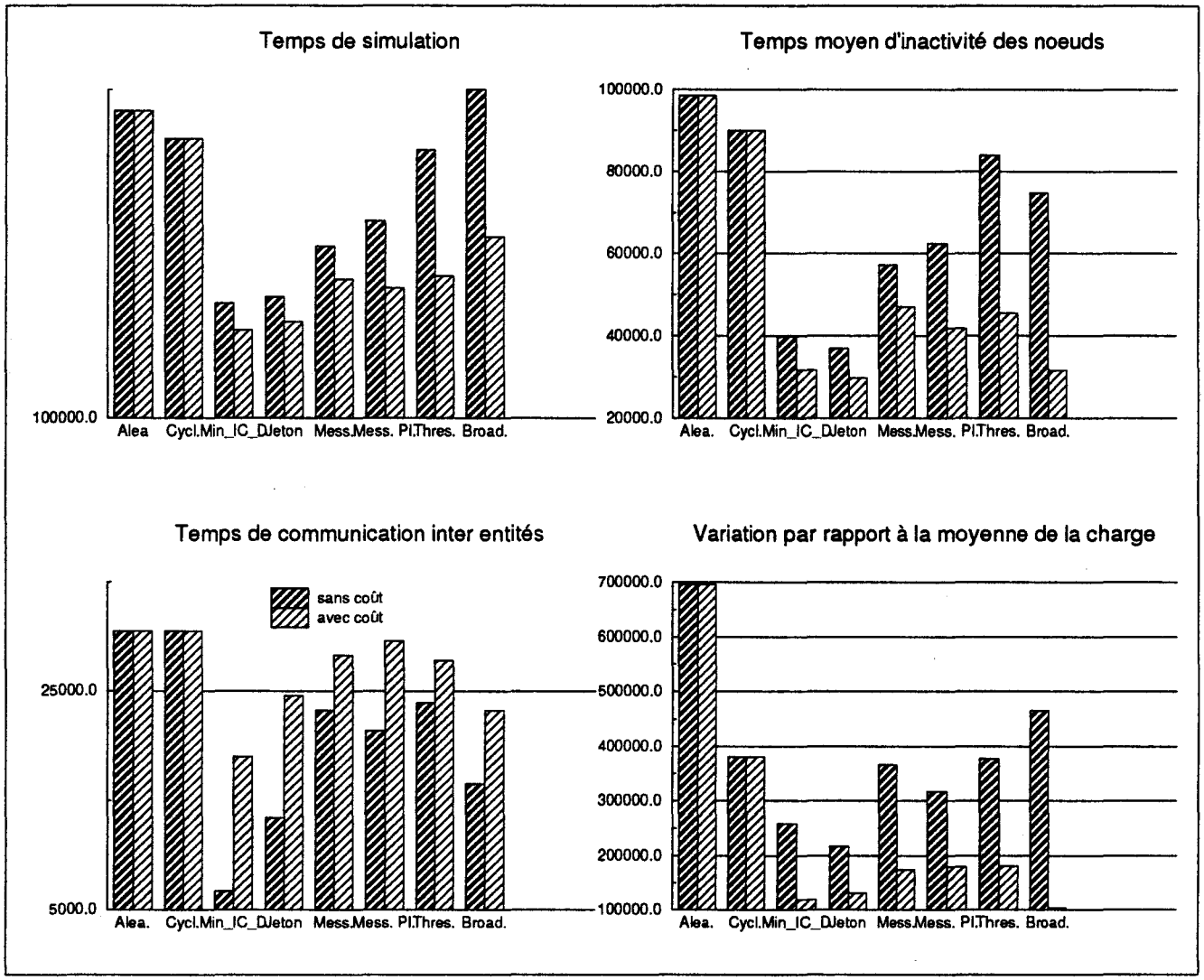


FIG. 4.5 - *Analyse des résultats de simulation de factorielle n modifiée (2)*

les colonnes les unes après les autres. Dans le cas où l'examen d'une colonne ne permet pas de placer une reine pour la recherche de solution en cours, cette solution est globalement rejetée.

Notre programme comporte quatre entités (voir le code en annexe B.3).

L'entité *root* lance l'application, et la recherche des solutions en parallèle à partir de la première colonne de gauche.

L'entité *reine* qui recherche la ou les position(s) valide(s) sur la colonne parcourue en fonction de la solution qui est en cours de construction.

L'entité *qlist* est un élément d'une liste qui constitue la solution en cours de construction. Une entité *qlist* représente la position d'une reine sur l'échiquer.

L'entité *sortie* est utilisée pour visualiser les solutions une fois qu'elles sont trouvées.

Le fonctionnement de l'algorithme est le suivant :

Au lancement l'entité *root* crée N entités *qlist* (N étant le nombre de lignes), une entité par ligne, et N entités *reine* associées à chacune des entités *qlist*. Les entités *reine* vont parcourir la colonne voisine de droite pour rechercher les positions valides en fonction de la position de l'élément *qlist*. Ce sont les entités *reine* qui vont effectuer le travail de recherche en parallèle et de manière indépendante. Lorsqu'une position valide est rencontrée, l'entité *qlist* associée est dupliquée, puis on crée une nouvelle entité contenant la position valide rencontrée que l'on va lier à l'entité *qlist* dupliquée. A la suite de quoi, une nouvelle entité *reine* est créée sur la colonne voisine de droite, elle est associée à la solution contenue dans la liste d'entités *qlist* précédemment créée. Quand une entité *reine* a terminé l'examen de la colonne, elle est détruite de même que la liste d'entités *qlist* associée. Lorsqu'aucune position valide n'a été trouvée, la solution en cours d'élaboration est abandonnée. L'algorithme se répéte jusqu'à ce qu'il y ait N entités *qlist* dans la liste, alors la référence du dernier élément de la liste qui pointe sur la solution complète est envoyée vers l'entité *sortie*.

Caractéristiques de l'application

Le taux de création d'instances d'entités *reine*, et *qlist* est important tout au long de l'exécution. L'entité *reine* communique essentiellement avec le dernier élément de la liste constituant la solution. Les éléments *qlist* communiquent principalement avec les autres éléments de la même liste. L'exécution ne conduit pas comme l'application factorielle à une structure régulière. Cette application définit le comportement générique de recherche d'un ensemble de solutions dans un espace construit dynamiquement, comme celui décrit par l'algorithme A^* en intelligence artificielle.

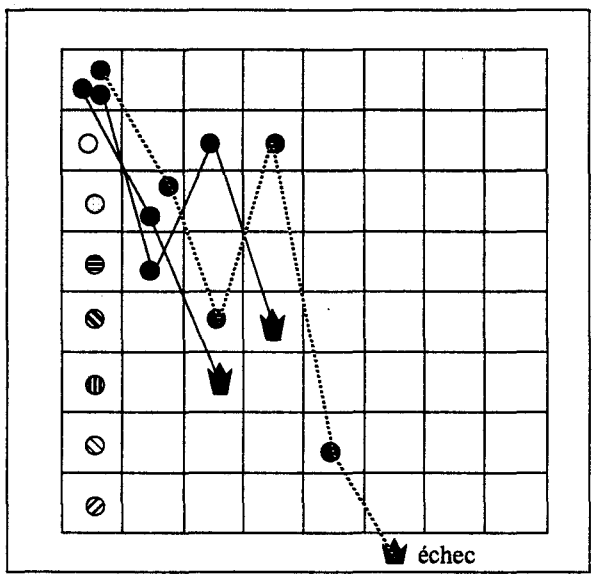


FIG. 4.6 - *Fonctionnement de l'algorithme des huit reines*

Evolution du temps de simulation en fonction de l'architecture

En utilisant les mêmes architectures que pour l'application factorielle nous obtenons les résultats présentés dans la figure 4.7.

On constate que les temps de simulation décroissent jusqu'à un nombre de sites proche de 25, ensuite plus le nombre de sites augmente et plus le temps augmente. La courbe des temps moyens d'inactivité des sites est linéairement croissante, cela signifie que la charge que représente l'application n'arrive pas à utiliser pleinement l'architecture disponible. En effet la durée de vie de chacune des entités est inférieure à la durée totale de l'exécution, le nombre d'instances d'entités existant simultanément est inférieure au nombre total d'entités créées, donc la charge instantanée de l'application n'arrive pas à alimenter en travail une architecture disposant d'un nombre important de sites.

Analyse des résultats de simulation de l'application *reines*

Nous avons simulé l'application *reines* sur une architecture avec 16 sites, organisée autour d'une topologie en forme de tore et en reprenant les paramètres globaux simulant le *MultiCluster-II*. Nous avons choisi, à la vue des résultats d'une série de simulations, l'indicateur de charge représentant le nombre d'entités présentes sur un site. Nous avons fait varier comme pour l'application factorielle les stratégies de placement et étudié le comportement de l'application.

Cette fois nous nous trouvons devant une application qui a un coût de communication aussi important que le coût de calcul, les entités sont donc le plus

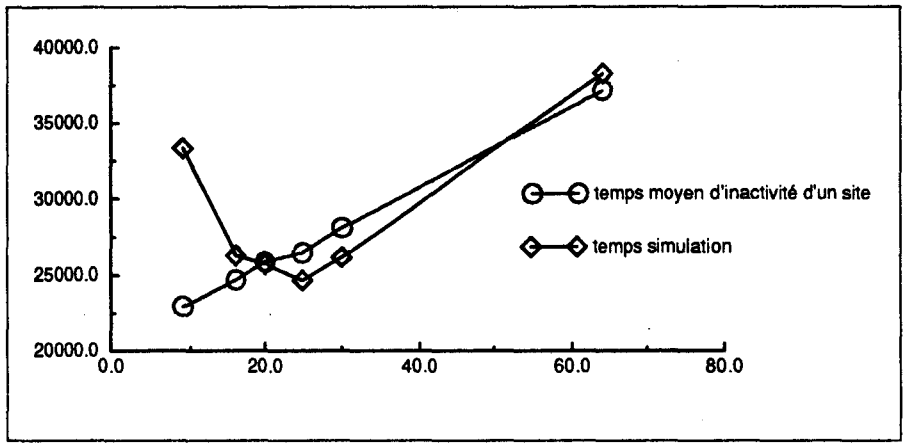


FIG. 4.7 - *Résultats pour reines en fonction de l'architecture*

souvent bloquées en attente d'un message. Le temps d'exécution de l'application est donc lié au coût de communication.

Le fonctionnement d'une liste veut que les éléments communiquent essentiellement avec leurs voisins et dans le cas de l'application avec l'instance d'une entité *reine* qui l'utilise. Pour réduire le coût de communication, l'algorithme de placement doit essayer de placer en priorité les éléments voisins d'une liste sur des sites voisins de l'architecture. Cela correspond à une particularisation du placement dirigée par l'application. Il est difficile d'introduire ce type de règle dans un algorithme de placement. Il est plus facile de laisser le programmeur exprimer des *souhaits* dans le source de l'application.

Dans le cas de l'application *reines*, le programmeur va indiquer qu'il souhaite placer les instances de l'entité *"élément d'une liste" qlist*, sur des sites voisins. Dans la figure 4.8 on donne les modifications à apporter au source pour particulariser le placement.

Le coût de communication peut aussi être réduit en donnant une valeur au facteur β plus importante dans la formule utilisée par l'algorithme de localisation. Cela aura comme résultat de privilégier une minimisation de la distance entre l'instance qui demande la création et l'instance créée devant l'examen de la charge des sites.

Les résultats comparatifs entre les différentes techniques utilisées sont présentés dans la figure 4.9 (Entités sans pseudo-calculs). On constate que la modification du source de l'application pour privilégier la localisation des entités qui doivent l'être, apporte une amélioration sur le temps d'exécution. Après détermination d'une valeur pour le facteur β, la plus appropriée possible, les améliorations observées sont moins significatives, et la modification du source donne parfois des temps de simulation plus élevés.

```
-- numero du site de l'instance qui demande la création
here := numero_site(self);
-- codage pour que l'algorithme de localisation place en priorité
-- sur un site voisin
site := f(here);
-- demande de creation d'une instance de qlist sur un site voisin
nql := creation (site) qlist (lig, col, nql);
```

FIG. 4.8 - *Modification du source de l'application pour un placement particularisé*

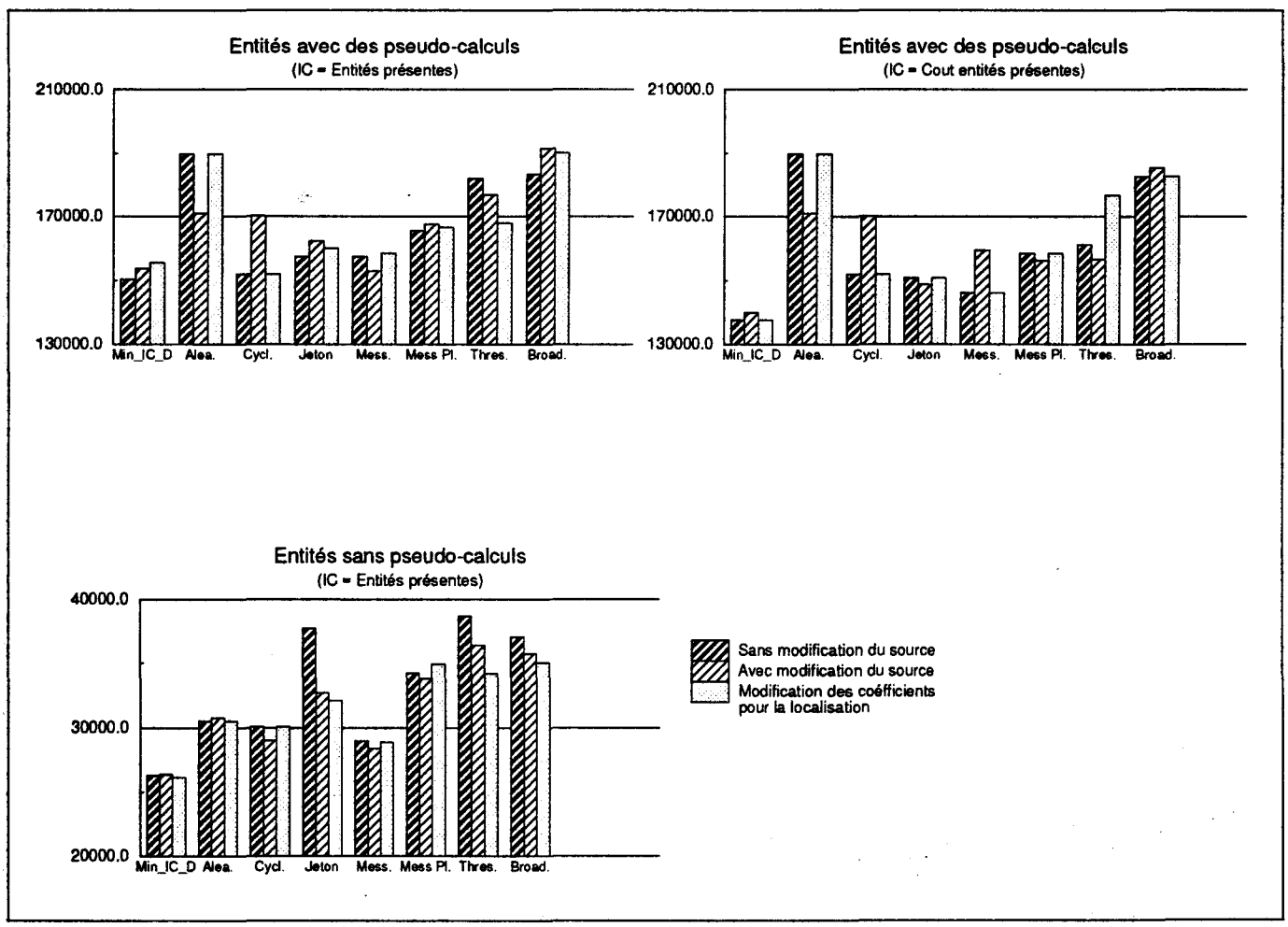


FIG. 4.9 - *Résultats de simulation pour "reines"*

Comme pour l'application factorielle nous avons testé le comportement de l'application lorsque, chaque instance d'entité demande un travail plus important en terme de calcul pur à la CPU, cela vis à vis des paramètres de l'architecture de la machine simulée.

Nous avons donc modifié le rapport qui était proche de 1:1 entre le coût de communication et le coût de calcul pour l'amener à une valeur de 1:6 en faveur du coût de calcul. L'entité *reine* utilisant six fois plus la CPU que l'entité *qlist* (rapport obtenu suite à une première simulation). La figure 4.9 (Entités avec pseudo-calculs) nous présente les résultats comparatifs avec les mêmes variantes que précédemment. Dans un premier graphique les résultats présentés sont ceux obtenus en utilisant un indicateur de charge comptabilisant sans distinction les entités présentes, et dans un deuxième graphique sont présentés les résultats d'un placement utilisant un indicateur de charge utilisant le facteur de coût estimé par entité.

On constate que la stratégie de placement en aveugle *Cyclique* donne de bons résultats. En effet la variance cumulée par rapport à une répartition idéale est meilleure en général que pour les autres stratégies utilisées. Ce phénomène s'explique par le fait que l'entité *qlist* est plus souvent créée que l'entité *reine*. Cela a pour effet de rendre négligeable la différence de coût qui existe entre les deux entités, ce qui nous ramène à une application comme factorielle qui se satisfait d'un placement uniforme.

Le placement différencié pour réduire le coût de communication se justifie un peu moins, étant donné que le rapport entre le coût de communication et coût de calcul a été modifié. Les résultats constatés sont d'ailleurs là pour le montrer. La même remarque peut s'appliquer pour la variation de la valeur du facteur β. L'utilisation de l'indicateur de charge tenant compte du coût des entités présentes apporte une amélioration bien plus significative. Pour le reste, les remarques faites sur les différentes politiques d'information utilisées restent vraies. Par exemple on constate que les instances d'entités sont mieux placées quand on utilise une politique d'information comme *Broadcast*, mais au prix d'une surcharge en coût de communication trop importante.

Conclusion sur le placement de l'application *reines*

L'application développe un espace de recherche équivalent à ceux obtenus par la recherche de solutions dans les algorithmes d'intelligence artificielle. Cet espace inconnu *a priori* est construit dynamiquement en créant des entités qui recherchent l'ensemble des solutions valides. Là où une recherche séquentielle aurait fait une opération de *backtrack*, pour envisager une autre voie, l'algorithme parallèle coupe la branche de recherche en arrêtant l'entité qui l'examine et poursuit la recherche sur les autres branches de manière indépendante.

L'implémentation de l'algorithme nous conduit à une exécution où la communication a un coût équivalent au coût de calcul. Dans ce cas les temps de communication qui disparaissent lorsque la charge calcul est importante par un phénomène de recouvrement, deviennent pénalisants pour le temps global d'exécution.

Nous avons donc proposé deux solutions pour le placement de l'application. Dans une première, le programmeur modifie le source de son application en introduisant des directives qui particularisent le placement de certaines instances d'entités. Dans une seconde solution nous modifions les facteurs de la formule de choix de la localisation en augmentant la valeur du facteur β relatif au coût de communication.

On constate que la première solution diminue le coût de communication de manière significative sans provoquer une trop grosse variation par rapport à la moyenne d'un équilibrage des charges parfait. La deuxième solution par contre, si elle diminue le coût de calcul tend à regrouper systématiquement les entités sur des sites voisins ce qui provoque une augmentation de la variance par rapport à un équilibrage parfait ce qui explique les mauvais résultats d'une telle solution.

L'ajout de pseudo-calcul dans le source des entités modifie le rapport entre coût d'exécution et coût de calcul. La communication devenant moins critique, il devient à nouveau primordial de bien équilibrer la charge. Ceci étant mieux fait en tenant compte des estimations de coût des entités dans l'indicateur de charge des sites.

Cette application nous a permis d'introduire la notion de placement multi-niveaux, le programmeur introduisant des directives de placement dans le source de l'application, en utilisant les résultats de simulation de son application lui indiquant les points où il fallait intervenir. L'algorithme de placement travaille cette fois avec les données sur l'état courant du système, les estimations de coût des entités, et les directives de l'utilisateur.

4.4.3 La multiplication de matrices

Le problème consiste à multiplier une matrice carrée A par une matrice carrée B. Notre implantation de la multiplication de matrice consiste à affecter à une instance de l'entité *wker*, une ligne de la matrice A, et de faire circuler les colonnes de la matrice B d'instance de *wker* en instance de *wker*. L'entité *wker* a la charge d'effectuer le calcul de chaque indice de la colonne de la matrice résultat (voir le code en annexe B.4).

Cette implantation de la multiplication de matrice est l'exemple qui montre que toutes les applications n'ont pas un taux de création de processus dynamique constant. En effet dans cet exemple, l'application ne fait des créations qu'au

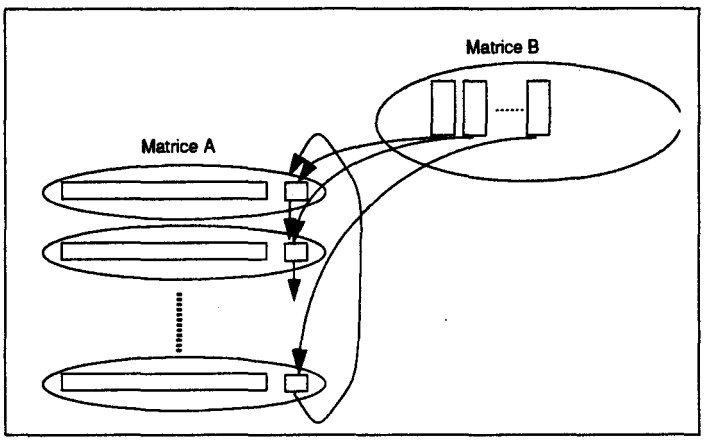


FIG. 4.10 - *La multiplication de matrices*

lancement de l'application, qui correspond à une période relativement courte par rapport au temps total de l'exécution.

De plus , tous les processus sont instance de la même entité *wker*. La figure 4.11 résume parfaitement l'analyse du placement. Seule la stratégie en aveugle *Cyclique* donne un temps de simulation inférieur aux autres stratégies. La raison est avant tout que, quelque soit la politique d'information utilisée, elle sera inefficace parce que les créations sont faites en un temps très court et dans une période réduite par rapport à la durée totale de la simulation. Le coût engendré par les échanges de messages est de surcroît inutile, puisqu'il n'y a plus de création par la suite. Ensuite les processus étant tous des instances d'une même entité (*wker*), la meilleure stratégie est celle qui applique un comportement équitable. Donc plus le coût d'implantation de la politique d'information est importante, plus le temps de simulation sera important.

Nous avons pu mettre en évidence dans cet exemple l'importance du placement dans le source de l'application des communications vis à vis des calculs. Dans la figure 4.11 on constate une diminution du temps de simulation simplement si on transmet la colonne vers l'instance *wker* avant d'effectuer les calculs plutôt qu'après. Cette diminution du temps de simulation n'apparaît que dans le cas d'un placement cyclique qui profite dans une plus large mesure de la réduction du temps moyen d'inactivité des noeuds, grâce au placement équitable.

En conclusion cette application aurait pû être placée statiquement. Cependant le placement dynamique en aveugle a l'avantage de s'adapter à une taille quelconque de la matrice et n'a pas besoin de connaître l'architecture de la machine.

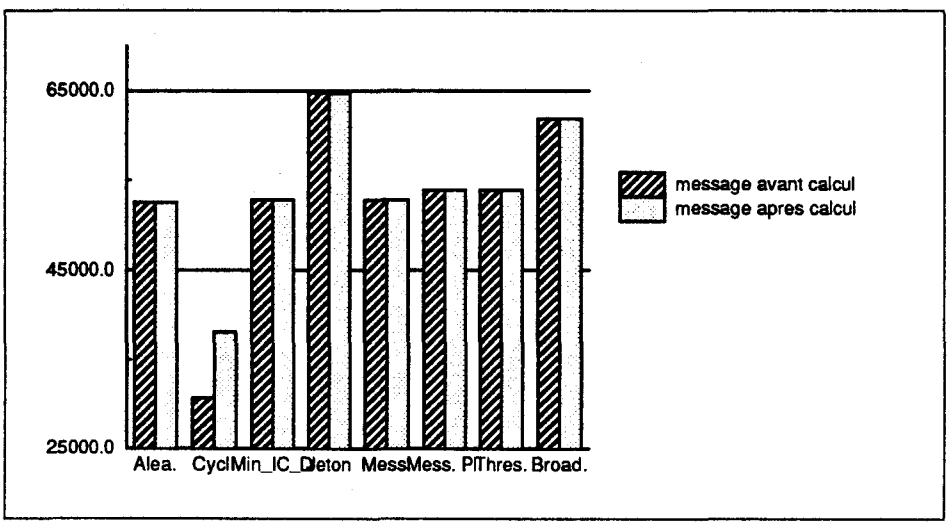


FIG. 4.11 - *Résultats de simulation pour la multiplication de matrice*

4.4.4 L'application Carwash

L'application *Carwash* simule l'activité d'une station de lavage et des clients qui l'utilisent. Cet exemple illustre l'adéquation entre le modèle de programmation par langage parallèle orienté objets et notre système basé sur les processus communicants. Les entités représentées sont :

- Les *voitures* qui sont clientes d'un service dans la station,
- Les *tronçons* qui sont utilisés par les voitures pour accéder à la station,
- Les *contrôleurs* qui sont responsables d'un ensemble de *postes de lavage*,
- Les *postes de lavage* qui rendent les services de la station de lavage. Nous avons créé plusieurs types de *postes de lavage*, chacun avec un coût de calcul particulier.

L'exécution de cette simulation d'une station de lavage se traduit de la manière suivante : Les *voitures* empruntent les *tronçons* pour arriver au niveau des *contrôleurs* qui gèrent les *postes de lavage*. Un *tronçon* ne peut être occupé que par une seule *voiture*. Un *contrôleur* gère un maximum de *n postes de lavage*. Les *postes de lavage* sont créés dynamiquement à la demande des *contrôleurs*. Ceux-ci vont déterminer le type du *poste de lavage* suivant les requêtes reçues de la part des *voitures*.

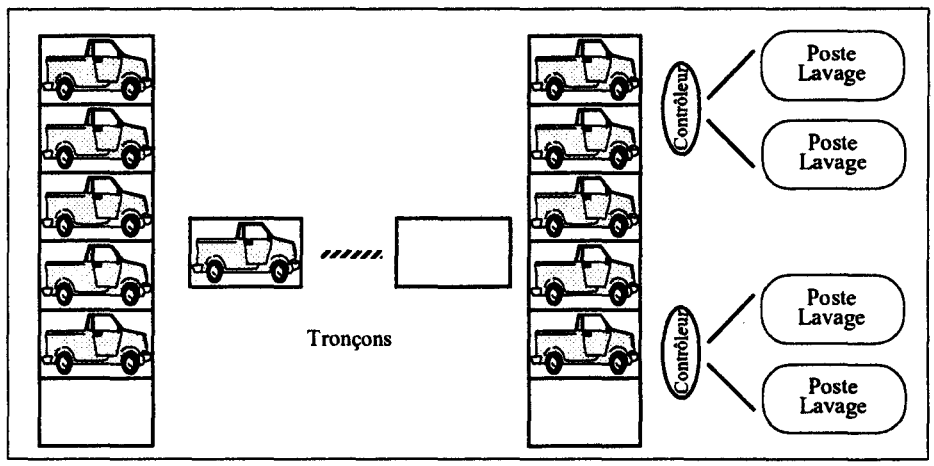


FIG. 4.12 - *L'application Carwash*

Caractéristiques de l'application

L'implantation de *Carwash* est basée sur une programmation par objets. Ce type d'application utilise des entités ayant des caractéristiques qui se rapprochent de celles présentées dans le chapitre 2. On y retrouve des gérants d'objets, des exécuteurs de méthode d'objet, et des gérants de structure de données. Dans *Carwash* les entités *voitures*, *tronçons*, et *contrôleurs*, peuvent être rangées dans la catégorie CGO (elles sont créées au lancement de l'application), tandis que les *postes de lavage* sont rangés dans la catégorie CEM (c'est le seul type d'entité qui est créée au cours de l'exécution).

Analyse des résultats de simulation de l'application *Carwash*

Le choix de l'indicateur de charge. Nous avons fait une série de simulations pour comparer les différents indicateurs de charge possibles. Le graphe (*f*) de la figure 4.13 en fait la synthèse. L'indicateur "*Ct Ent. Pres.*" donne le meilleur temps de simulation, nous l'avons donc retenu dans la suite des mesures. Etant donné que les entités ont des coûts variables, il faut tenir compte de leur coût respectif pour faire un bon placement dynamique. Le fait qu'il faille plutôt comptabiliser l'ensemble des entités présentes plutôt que les entités actives tend à montrer que pour cette application les entités sont rarement bloquées longtemps par des attentes de message.

Le choix de la politique de localisation. Nous avons testé quatre stratégies de localisation, deux fonctionnent en aveugle (*Aléatoire, Cyclique*), une autre (*Min_IC_Dist*) a déjà été utilisée dans les exemples précédents, nous y ajoutons la stratégie qui utilise pour le placement le type de l'entité (*Type Ent.*), méthode

présentée dans le paragraphe 3.2.6.5. Les résultats obtenus sont présentés dans la figure 4.13(a), la meilleure stratégie est *Min_IC_Dist*. On aurait pû s'attendre à obtenir un meilleur résultat pour la stratégie utilisant le type des entités, une grande partie des créations se faisant au lancement de l'application. Les bons résultats obtenus par *Min_IC_Dist* s'expliquent par l'utilisation du mécanisme d'anticipation du calcul de l'indicateur de charge sur le site qui demande la création qui est présenté dans le paragraphe 3.2.6.3. En effet les créations au lancement de l'application sont faites en majorité sur le même site qui dispose alors d'une bonne estimation de l'état de charge des autres sites.

Le choix de la politique d'information. Comme dans les exemples précédents nous avons testé les stratégies *Jeton, Message, Message Plus, Threshold, Broadcast*. Les résultats obtenus sont donnés dans la figure 4.13(b). Les stratégies *Message, Message Plus* se partagent les moins bons résultats. Ces stratégies sont victimes du bon fonctionnement de leur mécanisme d'échanges d'informations au lancement de l'application. En effet à chaque création, le message qui retourne la référence de l'entité modifie l'information de charge disponible du site origine. Malheureusement, cette valeur n'inclut pas l'entité créée (ou qui va être créée). Cela a pour conséquence de donner un placement mal équilibré au lancement de l'application. Ce mauvais placement au lancement de l'application entraine les mauvais résultats de simulation. Viennent ensuite les stratégies *Jeton, Broadcast*, alors que *Threshold* donne les temps de simulation les plus courts.

Les courbes de la figure 4.13(c) qui donnent les coût de communication engendrés par les politiques d'information, nous montre pourquoi la stratégie *Broadcast* donne un résultat moyen, alors qu'elle assure la répartition de charge la plus équitable (Fig. 4.13(e))

Quelque soit la stratégie de placement utilisée, les temps de simulation les plus courts sont obtenus avec une architecture composée de 16 sites. La figure 4.13(d) nous indique que plus le nombre de sites augmente et plus les sites sont inactifs, cela veut dire que la machine passe son temps à attendre des messages sans calcul à effectuer pour recouvrir ces attentes.

Conclusion sur le placement de "*Carwash*"

L'application est composée d'entités ayant des coûts et des types différents, ce qui implique que son placement doit être fait en tenant compte de ces différences. Son autre particularité est de faire une grande partie de ses créations au lancement de l'application, il faut donc garantir un placement initial qui les supporte. Enfin le coût d'exécution de l'ensemble de l'application ne nécessite pas une architecture dotée de nombreux sites.

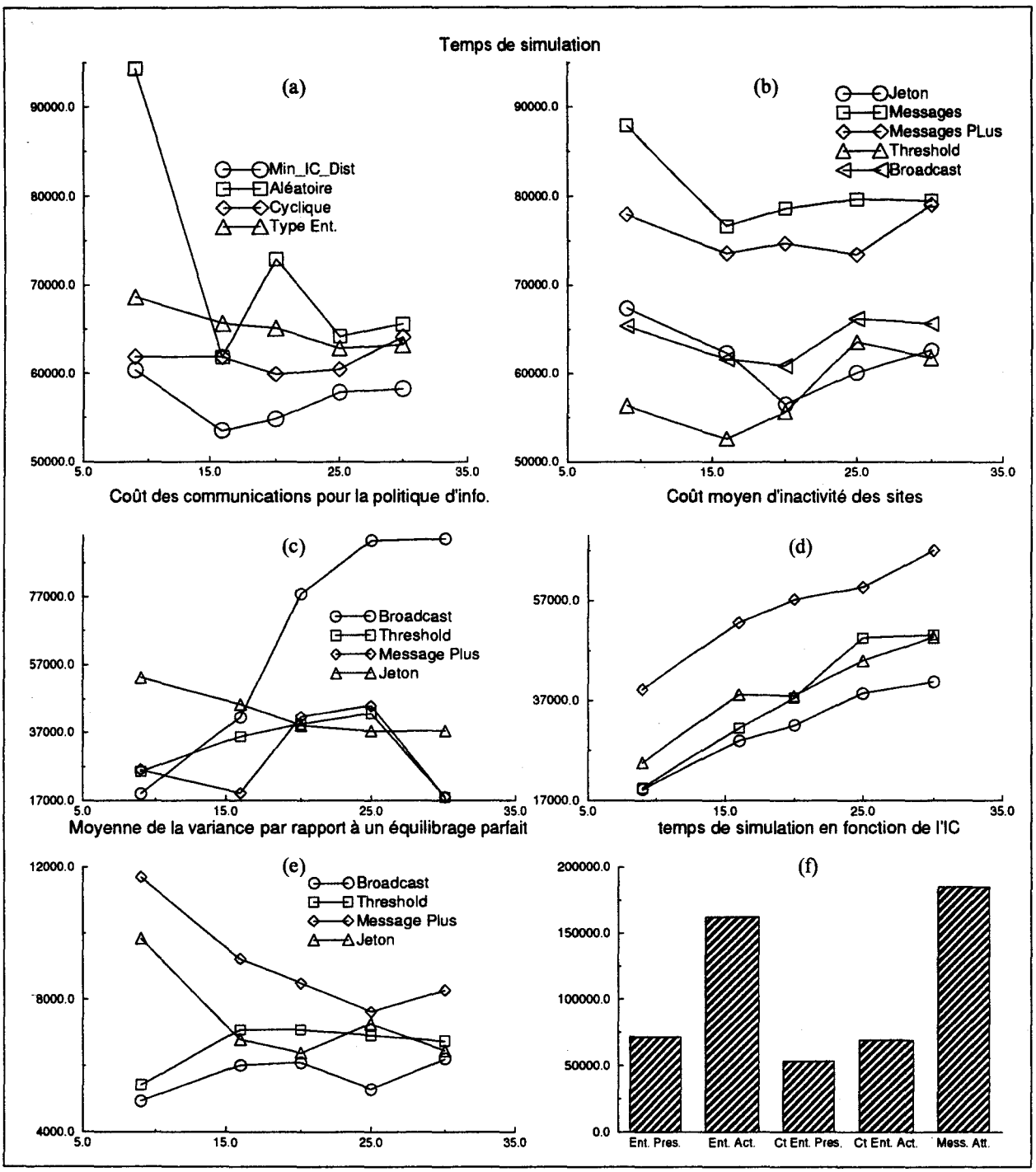


FIG. 4.13 - *Résultats de simulation pour "Carwash"*

4.4.5 Le crible d'Eratostène

Dans les exemples précédemment étudiés, nous avons toujours utilisé les mêmes valeurs pour les paramètres globaux de l'architecture. Seul le nombre de sites variait. Les résultats et les conclusions donnés sont contestables pour d'autres architectures, notamment celles qui n'introduisent pas la notion de distance comme les stations de travail connectées par un réseau local. Nous allons illustrer ce phénomène en utilisant l'application qui calcule le crible d'Eratostène présenté dans le paragraphe 3.2.3.2, et en modifiant les valeurs pour les paramètres globaux de deux types d'architecture :

- Des noeuds reliés par un réseau d'interconnexion;
- Un réseau de stations de travail.

Pour chacune de ces architectures nous allons modifier le coût unitaire de communication entre sites voisins, et comparer le comportement des stratégies de placement en fonction de cette variation.

La figure 4.14 donne une synthèse des résultats obtenus. Chaque graphique donne une suite de temps de simulation qui varient en fontion de la stratégie de placement dynamique utilisée. Les stratégies *Min_IC_D, Aléatoire, Cyclique, Jeton* ont déjà été utilisées précédemment, les deux dernières utilisent une politique de transfert qui tente de faire du partage de charge; c'est à dire que la création d'une entité sur un site distant n'est envisagée que si la charge locale est supérieure à une certaine valeur de seuil S.

Les graphiques *(a),(b),(c),(d)* de la figure 4.14 présentent l'utilisation de paramètres globaux pour l'architecture qui découlent des mesures faites sur le *MultiCluster-II*, auxquels nous avons fait varier le coût de communication. Dans le graphique *(a)* on s'aperçoit que les stratégies *Aléatoire* et *Cyclique* sont un peu moins bonnes que *Min_IC_D* et *Jeton*, alors que les stratégies qui font du partage de charge donnent de mauvais résultats. Dans les graphiques *(b),(c),(d)*, on obtient des résultats qui s'inversent, c'est à dire que *Aléatoire* et *Cyclique* donnent des résultats très mauvais en comparaison avec les autres stratégies. Plus le coût de communication devient important, et plus les stratégies qui font du partage de charge donnent de bons résultats. Ceci est dû à la diminution du nombre de communications inter-sites, ce qui réduit le temps de simulation.

Dans les graphiques *(e),(f),(g),(h)* de la figure 4.14 sont présentés les résulats de simulation suite à l'utilisation des paramètres globaux pour simuler un réseau de stations de travail, toujours avec une modification du coût de communication. Les remarques précédentes restent vraies dans des proportions moins flagrantes, en effet, dans cette architecture, on n'a plus la notion de distance qui intervient (chaque station est voisines des autres), les coûts de communication deviennent moins rapidement importants.

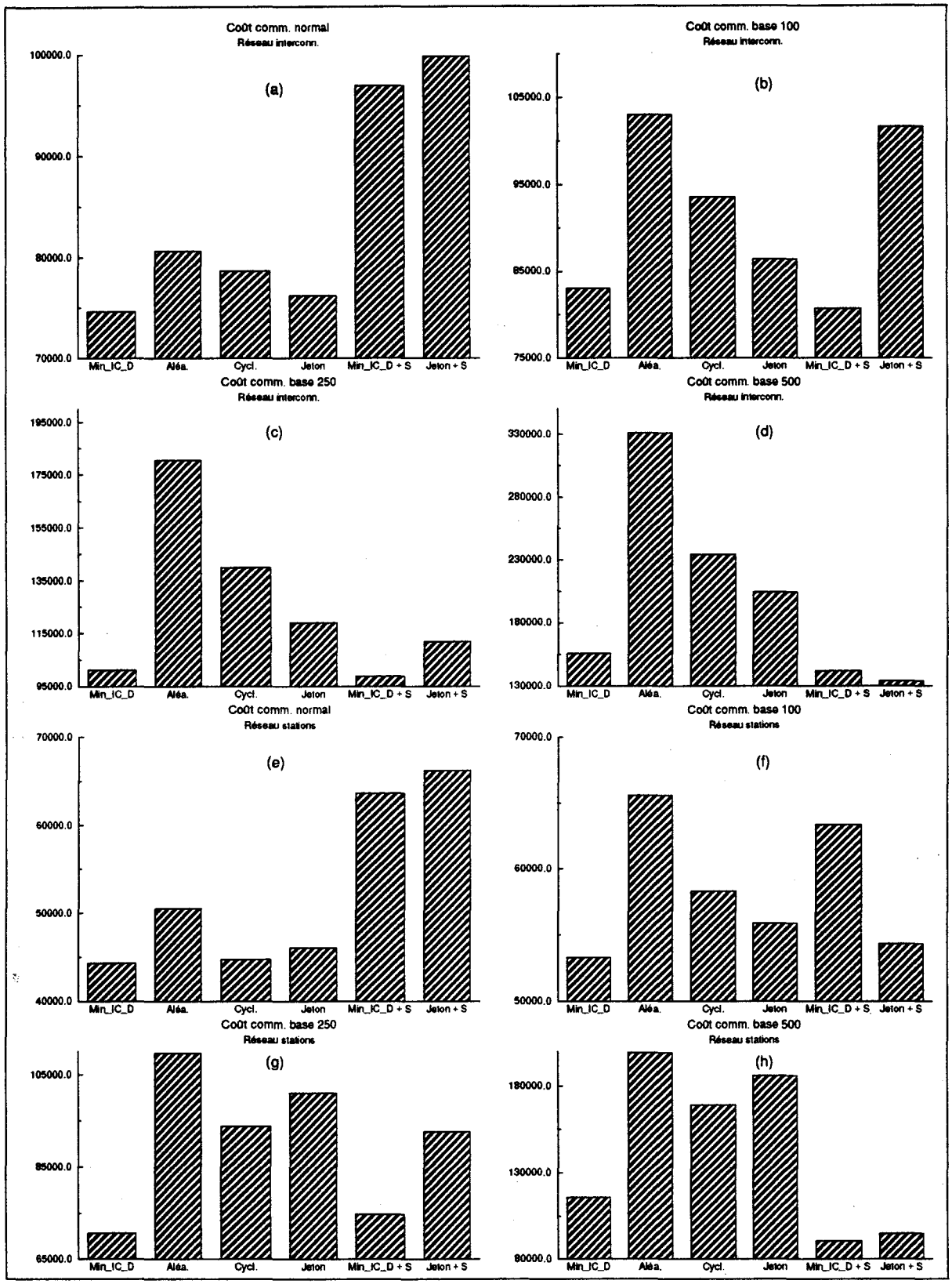


FIG. 4.14 - *Variation du coût de communication*

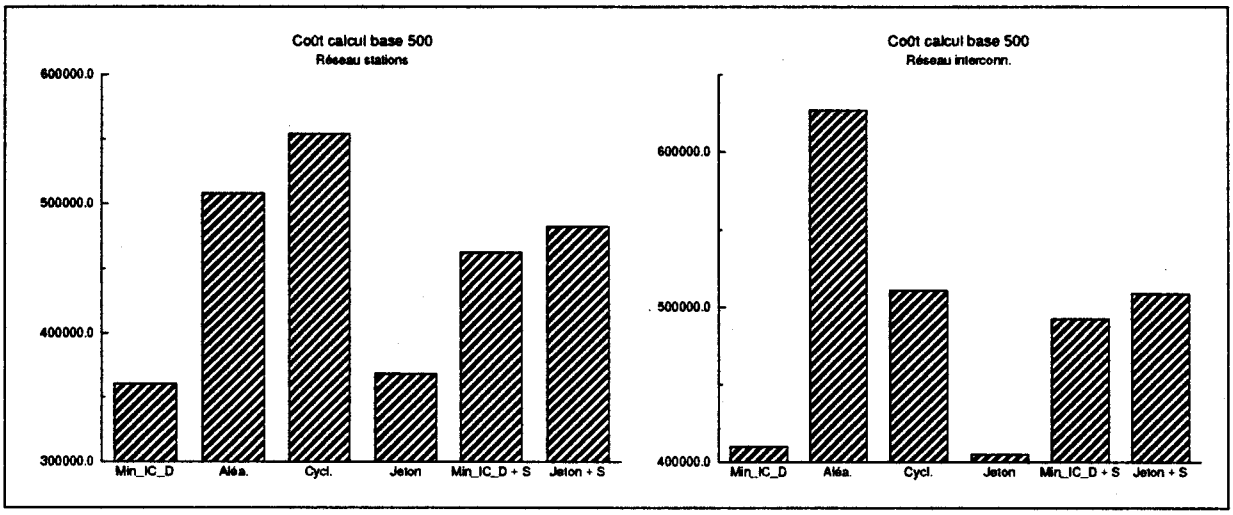


FIG. 4.15 - *Variation du coût de calcul*

Si plutôt que d'augmenter le coût de communication, on augmente le coût de calcul des entités *eratos*, les résultats (Voir Fig. 4.15) indiquent que la stratégie de partage de charge ne favorise pas une diminution du temps de simulation. La remarque pour les stratégies de placement en aveugle reste néanmoins toujours vraie, plus le coût de calcul augmente, et plus les résultats obtenus sont mauvais.

La variation du coût de communication, ou du coût de calcul ne modifie pas le classement relatif d'autres stratégies de placement également testées (*Broadcast, Message, Message Plus, Threshold*) pour cette application particulière.

4.5 synthèse sur l'aide à la détermination d'un stratégie de placement dynamique

Dans ce chapitre nous avons donné un échantillon des problèmes rencontrés lors de la détermination d'une stratégie de placement dynamique. Les acteurs qui conditionnent ce choix ont été clairement identifiés.

- L'architecture de la machine;
- L'objectif de la répartition;
- L'application.

Nous allons revenir sur chacun de ces acteurs pour préciser les caractéristiques qu'il faut retenir pour faire un choix de stratégie.

L'architecture de la machine

L'architecture de la machine conditionne le choix de la politique d'information, qui doit être utilisée. En effet, l'architecture influe sur l'*overhead* que va produire le mécanisme d'échange d'information de charge au cours de l'exécution. Les caractéristiques essentielles qui doivent être retenues sont :

- **Le nombre de sites et leurs caractéristiques**; plus ce nombre est grand, plus le coût de la mise en place d'une politique d'information précise sera important. Dans le cas où l'architectecture est hétérogène, la politique de localisation devra tenir compte des ressources particulières de chaque site. De même, le calcul de l'indicateur de charge devra intégrer une éventuelle différence de puissance de calcul.

- **Le type de réseau d'interconnexion**; dans le cas d'une machine parallèle, plus le diamètre de la topologie du réseau sera important, et plus le coût de la mise en place de la politique d'information sera important. La politique de localisation devra tenir compte de cette caractéristique pour placer au plus près les entités qui communiquent beaucoup. Dans le cas de stations connectées par un réseau local, celles-ci étant toutes voisines, cette caractéristiques n'entre plus comme paramètre dans la politique de localisation.

- **Le coût de communication**; cette caractéristique intervient dans le choix de la politique d'information, mais également sur la politique de transfert. Si le coût de communication est important, la politique de transfert devra limiter les communications inter-sites. La politique de localisation devra favoriser un placement sur des sites voisins pour les entités qui communiquent.

Les objectifs du placement dynamique

L'objectif que doit atteindre un placement dynamique peut varier d'une application à l'autre :

- Soit l'objectif est d'**obtenir un temps minimal d'exécution**; dans ce cas la préoccupation essentielle est de limiter au maximum l'*overhead* produit par la stratégie de placement dynamique et notamment de la politique d'information, cela tout en permettant un placement qui minimise le temps d'exécution. Une bonne approximation de la charge peut suffire à obtenir de bons résultats d'exécution.

- Soit l'objectif est d'**obtenir un temps minimal d'exécution, tout en garantissant les contraintes de l'application**; si l'application contraint le choix des sites pour certaines entités, la politique d'information et l'indicateur de charge doivent pouvoir garantir que les informations qui sont échangées vont permettre de satisfaire les contraintes. L'information de charge devient plus complexe et doit être plus précise, cela en essayant de limiter le coût de sa mise en place.

L'application

Grâce à notre plate-forme de simulation et notamment à une description aisée du comportement des applications à simuler par la génération d'une trace d'événements caractéristiques, nous avons pu faire ressortir des exemples présentés, et d'autres plus complexes, les caractéristiques importantes qui vont permettre un bon placement.

- **Le nombre d'entités distinctes**; si le nombre d'entités composant l'application est limité, alors le calcul de l'indicateur de charge n'aura pas besoin d'en tenir compte. Plus ce nombre sera petit et moins il sera nécessaire de mettre en place une politique d'information complexe, le placement dynamique pouvant même se satisfaire d'une politique de localisation en aveugle. Au contraire, si le nombre d'entités distinctes est important, l'indicateur de charge et la politique de localisation devront en tenir compte. En effet plus le nombre est important et plus on aura de chance de rencontrer des comportements variables qui induiront des coûts de calculs et des coûts de communications pouvant être très différents de l'un à l'autre. L'indicateur de charge devra alors intégrer le coût de chaque entité, dans certains cas la politique de localisation pourra s'adapter au type de l'entité à placer.
- **Le nombre de créations dynamiques et la variation de ce nombre au cours de l'exécution**; le choix de la stratégie de placement est conditionné par le nombre de créations dynamiques d'entités. Si ce nombre est faible, alors une politique d'information peu coûteuse conviendra. Si ce nombre est important il faudra utiliser une politique d'information performante et anticiper au maximum la modification de la charge, car dans certains cas aucune politique d'information n'aura la capacité de suivre l'évolution globale de la charge. La variation du nombre de création au cours de l'exécution est aussi une caractéristique importante. En effet, si toutes les créations sont faites au lancement de l'application, son placement relève plutôt d'une stratégie statique adaptée à l'architecture. Par contre si le nombre de créations se répartit sur le temps d'exécution ou sur certaines périodes de l'exécution, la politique d'information devra pouvoir s'adapter aux variations.

- **Le coût des entités**; si le grain des entités autrement dit la charge engendrée par chaque entité est faible par rapport aux autres grandeurs qui caractérisent l'architecture sur laquelle on les exécute, alors une politique de localisation en aveugle permettra d'obtenir un temps d'exécution minimal en comparaison avec tout autre mécanisme plus complexe. D'autre part, plus le coût des entités devient important et plus la politique de localisation doit en tenir compte.
- **Les relations entre les entités**; si le graphe des relations entre les entités a de nombreux arcs et que ces arcs ont un poids important, alors des politiques d'informations comme *Message* ou *Message Plus* permettront une bonne évaluation de l'état de charge global de la machine.

4.6 Conclusion

Dans ce chapitre, après une présentation de la méthode de détermination des paramètres globaux pour définir les architectures simulées par notre plate-forme, nous avons illustré notre propos par l'étude d'un ensemble d'applications caractéristiques et simples. Celles-ci nous ont permis de faire ressortir les caractéristiques essentielles des acteurs (architecture, objectif de placement, application) qui concourent à l'élaboration d'une stratégie de placement de processus actifs communicants.

Grâce à notre plate-forme de simulation, nous avons pu analyser finement le comportement des stratégies de placement implantées en faisant varier l'architecture sur lesquelles elles étaient utilisées. Nous avons pu tester leurs intérêts et leurs handicaps en fonction de l'application. Leurs comportements à l'exécution étant finement approchés grâce au langage de création d'une suite d'événements caractéristiques.

Conclusion et perspectives

Le projet PVC/BOX a défini une couche logicielle pour la modélisation et la construction d'applications parallèles basées sur l'approche objet. Cette couche logicielle utilise une structure unique (le Cac) pour définir les activités de l'application. Le Cac contient des données et un comportement. Une application se définit comme un ensemble de Cacs s'exécutant en parallèle. L'objectif de ce travail était de définir un outil de distribution des Cacs sur les noeuds de l'architecture parallèle, dans le but de réduire le temps de réponse. Nous avons abordé le problème en deux étapes de façon à répondre aux caractéristiques du modèle d'exécution.

Le regroupement des Cacs en Modules

La notion de module introduite par Luc Courtrai a été utilisée pour faire une première répartition de l'application. Le module est la structure visible par le système d'exploitation de la machine parallèle. Il contient le code des Cacs qu'il sera capable de prendre en charge au cours de l'exécution. Nous avons pris l'hypothèse qu'un seul module est chargé par noeud de l'architecture parallèle, cela signifie que la possibilité d'exécuter un Cac sur un noeud est conditionnée par la présence de son code dans le module chargé sur ce noeud.

Notre travail dans cette première étape, a été de déterminer le regroupement *idéal* du code des Cacs dans les modules. La notion d'*idéal* étant pour nous de réduire au maximum les défauts de localité afin de permettre une régulation plus efficace de la charge au cours de l'exécution. Ce cas idéal est obtenu quand le code de chaque Cac est dupliqué dans un nombre de modules égal au nombre de noeuds disponibles dans l'architecture (ce qui tend à se rapprocher du modèle

SPMD). Cependant, cette configuration ne peut pas toujours être atteinte. Le programmeur peut être amené à introduire des contraintes afin de diriger la répartition dynamique de la charge de façon à garantir par exemple le regroupement d'un sous-ensemble de données sur un noeud particulier. Le cas d'une architecture hétérogène introduit d'autres contraintes, comme la limitation en taille de la mémoire pour certains noeuds, ou l'absence d'une ressource indispensable à l'exécution d'un traitement particulier. Nous avons défini une fonction de coût permettant d'intégrer l'ensemble de ces contraintes sous forme d'une estimation du coût de la duplication (occupation mémoire, etc ...) et d'une estimation du coût d'exécution en fonction du noeud, mais aussi d'une estimation du coût de communication entre Cacs. Une minimalisation de cette fonction de coût ayant pour but de tendre vers la solution idéale présentée plus haut. Pour obtenir la solution minimale de la fonction de coût d'un regroupement nous avons utilisé l'algorithme heuristique du recuit simulé, qui est largement utilisé dans le cadre d'un placement statique.

La répartition dynamique des Cacs

Étant donné que le modèle utilisé se base sur des demandes explicites de création de Cac au cours de l'exécution, ainsi qu'une évolution dynamique des relations entre Cacs, une utilisation adaptée de l'architecture parallèle nécessite une répartition dynamique de la charge. Cette répartition dynamique s'inscrit dans un contexte où le nombre de demandes de création est important et cela pour des processus d'un grain fin ayant des comportements qui peuvent varier énormément de l'un à l'autre. Nous avons donc choisi de développer plusieurs stratégies de répartition dynamique, celles-ci sont intégrées dans le run-time PVC en ajoutant un Cac spécialisé dans chaque module de l'application. Elles ont comme caractéristiques communes: une distribution de l'algorithme de décision pour le choix de la localisation d'un processus sur chaque noeud de l'architecture, accompagnée d'une distribution plus ou moins complète de la représentation de l'état global de la machine. La localisation n'est pas remise en cause au cours de l'exécution du processus, le mécanisme de migration n'est donc pas utilisé.

L'utilisateur définit une stratégie de répartition en fonction de l'architecture de la machine dont il dispose, mais aussi en fonction des caractéristiques de l'application qu'il doit répartir. Les stratégies s'articulent autour de quatre composantes qui sont: 1) Une politique de représentation de l'état de la charge de la machine; 2) Une politique de transfert de charge; 3) Une politique d'échange d'information; et 4) Une politique de localisation.

Pour que l'utilisateur soit capable de faire le choix des paramètres qui vont régir la stratégie de répartition dynamique, nous avons mis en place une plateforme qui permet d'évaluer les stratégies de répartition dynamique. Cette plate-

forme repose sur trois descriptions: 1) Une description de l'application à répartir; 2) Une description de l'architecture cible; et 3) Une configuration des paramètres de la stratégie de répartition dynamique.

Le langage GENESE a été développé pour permettre une description de l'application et pour piloter le simulateur de notre plate-forme. Ce langage représente l'application par un ensemble d'entités, avec pour chaque entité, la description de son comportement sous forme du séquencement d'événements importants vis à vis de l'étude d'une répartition de charge. Ces événements sont au nombre de quatre: la demande de **création** d'une entité, l'envoi de **message**, l'**attente** d'un message, et le **calcul** local. Le programme GENESE est interprété par notre simulateur et génère une suite d'événements que nous utilisons pour analyser le comportement de l'application.

La description de l'architecture permet dans un premier niveau de définir chaque noeud de la machine. Dans un deuxième niveau la topologie du réseau d'interconnexion est définie. Enfin un dernier niveau permet de donner un coût relatif de l'exécution de chaque événement tracé.

La description de la stratégie de répartition dynamique reprend chacune des composantes introduites plus haut.

Les résultats obtenus avec la plate-forme d'évaluation nous ont permis de préciser les critères de sélection d'une stratégie de répartition dynamique. Pour l'architecture de la machine les critères importants sont :

- Le nombre de noeuds;
- Le caractère homogène ou hétérogène de l'architecture;
- La topologie du réseau d'interconnexion;
- Le coût des communications relativement à la puissance des noeuds.

Pour l'application les critères importants sont :

- Le nombre d'entités distinctes et leurs coûts respectifs;
- Le nombre de créations dynamiques et la variation de ce nombre au cours de l'exécution;
- Les relations entre les entités.

Perspectives

Le domaine de la distribution de charge sur une architecture décentralisée est en pleine évolution et cela du fait de l'introduction sur le marché de nouvelles

architectures de machine parallèle citons par exemple la *Farm-ALPHA* de chez *Digital*, la *SP-1* d'*IBM*, la *CS-2* du consortium *PCI*, la *Paragon* d'*Intel*, ou encore la *CM-5* de *TMC*. L'orientation de ces machines est de proposer un système d'exploitation permettant un accès multi-utilisateurs pouvant exécuter plusieurs applications de manières concurrentes. Cette vision d'une machine parallèle se rapproche d'un environnement de stations de travail reliées par un réseau d'interconnexion très rapide, et mettant en place des mécanismes comme la mémoire virtuellement partagée, donnant l'illusion à l'utilisateur de disposer d'une super station de travail. Le problème de la répartition de charge devra donc intégrer le facteur multi-applications utilisant une granularité variable.

Les perspectives à court terme sont d'une part l'amélioration de la plate-forme d'évaluation à différents niveaux :

- Description plus fine du réseau d'interconnexion qui permettra d'introduire des liaisons hétérogènes et un coût de communication différencié.
- Ajustement du calcul de coût des communications en prenant en compte le trafic et la charge des sites.
- Ajout d'écrans de visualisation pour l'aide au prototypage et l'étude des performances à l'aide des traces produites en se rapprochant de l'outil ParaGraph[HE91].

D'autre part nos perspectives s'orientent vers l'implantation du run-time PVC sur une architecture de type *Farm-ALPHA*,qui intégrera un mécanisme de régulation dynamique multi-grains utilisant une couche communication basée sur l'approche PVM ou MPI, et permettra d'introduire l'aspect hétérogène. Cela permettra aussi de disposer d'un outil d'évaluation de la charge qui soit multi-applications et qui facilitera le travail des autres composantes de la stratégie de répartition.

Les perspectives à plus long terme sont de définir un catalogue de critères réutilisables de placement d'un processus dans un environnement décentralisé. Ce catalogue pouvant définir les bases d'un langage de spécification des ressources utilisées par un processus. Cette spécification est ensuite utilisée pour aider au travail de la répartition [Win92]. Un placement statique peut être envisagé en prenant comme hypothèse de travail que la machine est dans un état de charge représentatif d'une situation normale d'utilisation. Le résultat du placement statique ne se contentera plus de définir un choix unique de localisation par processus, mais proposera une liste de choix permettant d'adapter le placement à l'état courant de la machine. Un algorithme de répartition dynamique prendra alors en charge le choix définitif. Le langage de spécification peut aussi être considéré comme une liste de fait et de règles d'inférences qui sera utilisée par un algorithme de répartition dynamique utilisant un système expert.

Dans un avenir proche on peut présager que les applications parallèles seront exécutées dans un réseau de machines hétérogènes. L'architecture de chaque machine pourra être une station de travail classique, ou une machine parallèle. La notion de machine dédiée va disparaître pour s'orienter vers une utilisation multi-applications et multi-utilisateurs. Dans ce contexte les systèmes d'exploitation vont devoir intégrer la notion de partage de ressources dans un espace fédéré par le réseau d'interconnexion. Cette évolution montre l'importance d'un traitement efficace de l'équilibrage de la charge et l'intérêt de développer une normalisation de la spécification des besoins particuliers des applications parallèles et les ressources disponibles (avec leurs évolutions dynamique) de chaque machine.

Annexe A

Syntaxe du langage GENESE

```
program:          PROGRAM IDENT liste_decl_var_op liste_entite

liste_decl_var_op: /* vide */ | liste_decl_var

liste_decl_var: VAR decl_vars ';'

decl_vars:        decl_var | decl_var ',' decl_vars

decl_var:         IDENT decl_ind

decl_ind:         /* vide */ | '[' NUM ']'

liste_entite:     entite ';' liste_entite | entite ';'

entite:          IDENT '(' liste_par_ent ')' '{' entete_ent bloc '}'

liste_par_ent:          paras_entite
                      | paras_entite ',' liste_par_ent
                      | /* vide */

paras_entite:     IDENT

entete_ent    :   liste_entete
```

```
liste_entete:   liste_entete ';' element_entete | element_entete

element_entete:   taille
                | domaine_entite
                | connexion_entite
                | liste_decl_var
                | contraintes
                | type
                | charge_c
                | duree_vie
                | coupure


taille:         TAILLE ':' '=' NUM

domaine_entite: DOMAINE ':' '=' liste_site

liste_site:   nom_site
                | nom_site ',' liste_site
                | /* vide */

nom_site:       NUM

connexion_entite: CONNEX ':' '=' liste_ent_cont

liste_ent_cont:   entite_cont | entite_cont ',' liste_ent_cont

entite_cont: IDENT '(' NUM ')'

contraintes: CONTRAINTE ':' '=' liste_contraintes

liste_contraintes: liste_contraintes ',' contrainte | contrainte

contrainte:   containte_i_ent
                | contrainte_ent_site
                | contrainte_qt_site

containte_i_ent: WITH IDENT | WITHOUT IDENT

contrainte_ent_site: NEED IDENT

contrainte_qt_site: TIME NUM
```

```
type :          TYPE ':''=' IDENT

charge_c :      CHARGE_C ':''=' NUM

duree_vie :     DUREE ':''=' NUM

coupure :       COUPURE ':''=' NUM

var:            IDENT index

index:          /* vide */ | '[' expr ']'

bloc:           BEGIN liste_instructions END


mquad:          /* vide */

stmt: instruction ';' | instruction_if

liste_instructions: liste_instructions  mquad stmt | stmt

instruction:  /* vide */
              | bloc
              | instruction_tq
              | instruction_gen
              | affectation


avant_else:     /* vide */

instruction_if: IF  expr_bool  THEN mquad stmt

expr:             expr '+' terme
              | expr '-' terme
              | '-' terme | terme

terme:            terme '*' facteur
              | terme '/' facteur
              | facteur

facteur:        var | NUM | SELF |'(' expr ')'

expr_bool:        expr_bool '|' '|' mquad expr_bool_d
```

```
                | expr_bool '&' '&' mquad expr_bool_d
                | '!' expr_bool_d
                | expr_bool_d

expr_bool_d: '(' expr_bool ')' | '(' expr OPREL expr ')'

instruction_tq: TQ mquad  expr_bool  FAIRE mquad instruction

affectation:    var ':' '=' instruction_gen | var ':' '=' expr

instruction_gen:  creation
                | message
                | attente
                | calcul
                | fin
                | autre

creation:       CREATION algorithme IDENT pars

algorithme:     '(' var ')' | '(' NUM ')' | /* vide */

pars:           /* vide */ | '(' liste_paras ')'

liste_paras:    par | liste_paras ',' par | /* vide */

par:            var | SELF | NUM

autre:            RANDOM pars
                | PRTSTR '(' STRING ')'
                | PRTINT '(' par ')'
                | PRTNL
                | EXTERNE pars

message: MESSAGE var_op var pars

calcul: CALCUL var_op
attente: ATTENTE var_op pars
var_op: '[' par ']' | /* vide */
fin: FIN
```

Annexe B

Exemples de programmes GENESE

B.1 Le crible d'Eratostène

```
------------------------------------------------------------------------
--
-- Exemple du crible d'ERATOSTENE
--
------------------------------------------------------------------------
program eratos
var limite, cal;
root()
{
  taille := 5000;
  type := CRO;
  domaine := 0;

  var i, eratos2, rqt;

  begin
    printstring("Lancement de ERATOS");
    printnl;
    limite := 2000;
    i := 2;
    eratos2 := creation eratos (self,self,i);
```

```
    i := 3;
    rqt := -1;
    tantque (i < limite) faire
    begin
      message eratos2 (i, rqt);
      i := i + 1;
    end;
    i := -1;
    message eratos2 (i, rqt);
    attente;
    rqt := -2;
    i := 0;
    message eratos2 (i, rqt);
    attente;
    printnl;
    fin;
  end
};

eratos(racine,pere,valeur)
{
  taille := 1000;
  type := CEM;

  var donnee, rqt, next, fini, q, r,cal;

  begin
    printint(valeur);
    printstring(" ,");
    cal := 100;
    fini := 0;
    next := -1;
    tantque (fini == 0) faire
    begin
      attente (donnee, rqt);
      if (rqt == -1) then
      begin
        if (donnee > 0) then
        begin
          q := donnee / valeur;
          r := donnee - (q * valeur);
          calcul [cal];
          if (r != 0) then
            if ( next == -1) then
              next := creation eratos (racine,self, donnee);
```

```
            else
              message next (donnee, rqt);
        end;
        else  -- donnee <= 0
        begin
          if (next == -1) then
            message racine ();
          else
            message next (donnee, rqt);
        end;
      end;
      else  -- rqt == -2
      begin
        if (next == -1) then
          message racine ();
        else
          message next (donnee, rqt);
        fini := 1;
      end;
    end;
    fin;
  end
};
```

B.2 Factorielle n

```
------------------------------------------------------------------------
--
--    C A L C U L   D E   F A C T O R I E L L E   N
--
------------------------------------------------------------------------
program fac
var limite;
root()
{
  taille := 5000;
  type := CRO;

  var i, base;

  begin
    limite := 200;
--    cal := 1000;
    base := 1;
```

```
    i := creation fac (self,base,limite);
    attente (i);
    fin;
  end
};

fac(pere,a,b)
{
  taille := 1000;
  charge_calcul := 1;
  type := CEM;

  var para, para1, here,cal;

  begin
    if (a == b) then
      message pere (a);
    else
      begin
        para := (a+b)/2;
        para1 := ((a+b)/2 ) + 1;
        cal := 10;
        calcul [cal];
        here := -1;
        creation (here) fac (self,a, para);
--   Creation de fac sur un voisin
--        here := numero_site(self);
--        here := -(2 - here);
        creation (here) fac (self,para1, b);
        attente (para);
        attente (para1);
        para := para + para1;
        message pere (para);
      end;
    fin;
  end
};
```

B.3 Les huit reines

```
-- Programme qui recherche l'ensemble des combinaisons possibles
-- de placement des N reines sur un echiquier NxN
```

```
program reines
var N, nb_reines;

-- Lancement de l'application

root()
{
  taille := 5000;
  type := CRO;
  charge_calcul := 1;

  var i, ql, s, next;

  begin
    N := 6;
    nb_reines := 0;
    next := -1;
    s := creation sortie (self);
    i := 1;
    tantque (i <= N) faire
    begin
      ql := creation qlist (i,1,next);
      creation reine (ql,2,s);
      nb_reines := nb_reines + 1;
      i := i + 1;
    end;
    attente [s] (i);
    if (i == 0) then
      printstring("Pas de solutions");
    fin;
  end
};



reine(mql,col,s)
{

  type := CEM;
  charge_calcul := 6;
var lig, bool, nql, ncol, next, cal, site, here;

begin
  ncol := col + 1;
```

```
  lig := 1;
  cal := 1000;
  tantque (lig <= N) faire
  begin
    calcul [cal];
    message mql (1, lig, col, self);
    attente  (bool);
    if (bool == 1) then
    begin
      message mql (2, 0, 0, self);
      attente [mql] (nql);
--      here := numero_site(self);
--      site := - (here + 2);
      site := -1;
      nql := creation (site) qlist (lig, col, nql);
      if (col == N) then
        message s (1, nql);
      else
      begin
        creation reine (nql, ncol, s);
        nb_reines := nb_reines + 1;
      end;
    end;
    lig := lig + 1;
  end;
  message mql (4, 0, 0, 0);
  nb_reines := nb_reines - 1;
  next := -1;
  if (nb_reines == 0) then
    message s (2, next);
  fin;
end
};

qlist (lig, col, next)
{

  type := CEM;
  charge_calcul := 1;
var ok, rqt, l, c, demandeur, nql, n, difflig, diffcol, here, site, cal;

begin
  ok := 1;
  cal := 400;
  tantque (ok == 1) faire
```

```
  begin
    attente (rqt, l, c, demandeur);
    if (rqt == 1) then -- check
    begin
      difflig := l - lig;
      if (difflig < 0) then
        diffcol := col - c;
      else
        diffcol := c - col;
      calcul [cal];
      if ((difflig == 0) || (difflig == diffcol)) then
        message demandeur (0);
      else
      begin
        if (next == -1) then
          message demandeur (1);
        else
          message next (rqt, l, c, demandeur);
      end;
    end;
    if (rqt == 2) then -- copy
    begin
      if (next == -1) then
      begin
        nql := creation qlist (lig, col, next);
        message demandeur (nql);
      end;
      else
      begin
        message next (2, l, c, self);
        attente [next] (n);
--        here := numero_site(n);
--        site := - (2  + here);
        site := -1;
        nql := creation (site) qlist (lig, col, n);
        message demandeur (nql);
      end;
    end;
    if (rqt == 3) then -- get
    begin
      message demandeur (lig, col, next);
    end;
    if (rqt == 4) then -- fin
    begin
      if (next != -1) then
```

```
        message next (4, 0, 0, 0);
      ok := 0;
    end;
  end;
  fin;
end
};


sortie(pere)
{

  charge_calcul := 1;
var ok, i, lig, col, rqt, ql, oql, nb_sol;

begin
  ok := 1;
  nb_sol := 1;
  tantque (ok == 1) faire
  begin
    attente (rqt, ql);
    if (rqt == 1) then
    begin
      i := N;
      printstring("Solution ");
      printint(nb_sol);
      oql := ql;
      tantque (i > 0) faire
      begin
        message ql (3, 0, 0, self);
        attente [ql] (lig, col, ql);
        printstring(" R");
        printint(i);
        printstring(" -> ");
        printint(lig);
        i := i - 1;
      end;
      printnl;
      nb_sol := nb_sol + 1;
      message oql (4, 0, 0, 0);
    end;
    if (rqt == 2) then
    begin
      message pere (nb_sol);
      ok := 0;
```

```
    end;
  end;
  fin;
end
};
```

B.4 La multiplication de matrice

```
-- Multiplication de matrices reparties par lignes et colonnes
-- matA * matB = matR

program multimat

  var wker[1024];

root()
{
  taille := 5000;
  type := CRO;

  var i, base, limite, j, t, k;

  begin
    limite := 32;


    -- Creation des workers qui vont faire le calcul un par ligne

    j := 0;
    tantque (j < limite) faire
    begin
      wker[j] := creation worker(self, limite, j);
      j := j + 1;
    end;

    -- envoi des adresses de successeur aux wkers

    j := 0;
    tantque (j < limite) faire
    begin
      k := j + 1;
      if ( j != (limite - 1)) then
        message wker[j] (wker[k], k);
```

```
      else
        message wker[j] (wker[0], 0);
      j := j + 1;
    end;

    -- envoi des colonnes de la matrice B

    t := 5;
    j := 0;
    tantque (j < limite) faire
    begin
      message [t] wker[j] (j); -- envoi de la colonne j vers le wker j
      j := j + 1;
    end;

    -- Attente du resultat

    j := 0;
    tantque (j < limite) faire
    begin
      attente (i);
      j := j + 1;
    end;
    fin;
  end
};


-- entite worker contient une ligne de la matrice A
-- il attend une colonne de la matrice B fait son calcul avec
-- et la donne au successeur si il y en a un

worker(pere, max, nl)
{
  taille := CEM;
  taille := 1000;

  var nc, nbc, j, cal, succ, nsucc, t;

  begin

    t := 5;
    nbc := 0;
    cal := 10;
```

```
    attente [pere] (succ, nsucc); -- attente de l'adresse du succ et du numero

    tantque (nbc < max) faire
    begin
      attente (nc);

      if (nsucc != nc) then
        message [t] succ (nc);
      nbc := nbc + 1;

      j := 0;
      tantque (j < max) faire
      begin
        calcul [cal]; -- correspond au coût de calcul de la multiplication
        j := j + 1;
      end;
    end;

    message pere (nl);
    fin;
  end
};
```

B.5 L'application *Carwash*

```
------------------------------------------------------------------------
--
--    C A R W A S H
--
------------------------------------------------------------------------
program carwash
var etat_global, cpt_ch;
root()
{
  type := CRO;
  charge_calcul := 1;

  var nb_tronc, nb_voit, nb_ctrl, nb_ptlav, adr_tronc_f, adr_tronc_c,
adr_tronc_d, tab_ctrl[10], tab_tronc[50], i, j, null, l, voit, ctrl, n,
tronc_crt;

  begin
    printstring("Lancement de carwash : ");
```

```
    printnl;
    nb_tronc := 1;
    nb_ctrl := 3;
    cpt_ch := 0;
    nb_ptlav := 2;
    nb_voit := 50;
    etat_global := 0;
    null := -1;

-- Creation des troncons de sortie et avant les controleurs

    adr_tronc_f := creation troncon (self, 100, null, 1);
    adr_tronc_c := creation troncon (self, 100, null, 2);

-- Creation des troncons qui conduisent au troncon avant les controleurs

    i := 0;
    tronc_crt := adr_tronc_c;
    tantque ( i < nb_tronc) faire
    begin
      tab_tronc[i] := creation troncon(self, 1, tronc_crt, 1);
      tronc_crt := tab_tronc[i];
      i := i + 1;
    end;

    adr_tronc_d := tronc_crt;

-- Creation des controleurs

    i := 0;
    tantque ( i < nb_ctrl) faire
    begin
      tab_ctrl[i] := creation controleur ( self, adr_tronc_c, adr_tronc_f, nb_ptlav);
      i := i + 1;
    end;

-- Creation des voitures

    i := 0;
    tantque ( i < nb_voit) faire
    begin
      creation voiture (self, adr_tronc_d, i);
      i := i + 1;
    end;
```

```
-- Attente de la sortie des voitures de la station de lavage

    i := 0;
    printstring("Les voitures : ");
    tantque ( i < nb_voit) faire
    begin
      attente (voit, n);
      printint(n);
      printstring(" ,");
      i := i + 1;
    end;
    printstring(" sont propres");
    printnl;

-- Fin de l'application

    etat_global := 1;

-- Attente fin des controleurs

    i := 0;
    tantque ( i < nb_ctrl) faire
    begin
      attente (ctrl);

      i := i + 1;
    end;

-- Message d'arret des troncons

    message adr_tronc_c (self, 5);
    message adr_tronc_f (self, 5);
    i := 0;
    tantque ( i < nb_tronc) faire
    begin
      message tab_tronc[i] (self, 5);
      i := i + 1;
    end;

    printstring("Fin de l'application");
    printnl;
    fin;
  end
};
```

```
troncon(pere,max,next,typ)
{
  type := CRO;
  charge_calcul := 10;

  var fini, cpt, rqt, adr_next, i, j, k ,dem, rep, null,park[100],etat[100];

  begin
    fini := 0;
    cpt := 0;
    null := -1;
    i := 0;
    tantque ( i < max ) faire
    begin
      park[i] := -1;
      etat[i] := 0;
      i := i + 1;
    end;
    tantque (fini == 0) faire
    begin
      attente (dem, rqt);

-- Message de demande pour entrer dans le troncon

      if (rqt == 1) then
      begin
        if ( cpt < max) then
        begin
          i := 0;
          tantque (etat[i] == 1) faire -- recherche une place libre
            i := i + 1;
          park[i] := dem;  -- met la voiture sur la 1ere place libre
          etat[i] := 1;
          cpt := cpt + 1; -- ajoute la voiture sur le troncon
          rep := typ;
          adr_next := next;
        end;
        else
        begin
          rep := 0;
          adr_next := -1;
        end;
        message dem (rep, adr_next);
      end;
```

```
-- Message de sortie d'une voiture du troncon

      if ( rqt == 2) then
      begin
        i := 0;
        tantque ((park[i] != dem) && (i < max)) faire
          i := i + 1; -- recherche de la voiture dem
        park[i] := -1;
        etat[i] := 0;
        cpt := cpt - 1;
      end;

-- Message d'un controleur

      if (rqt == 3) then
      begin
        if (cpt > 0) then -- y'a des voitures en attente
        begin
          k := random (cpt); -- choix au hasard d'une voiture
          i := 0;
          j := 0;


          tantque ((etat[i] == 0) && (i < max)) faire
            i := i + 1; -- recherche la 1ere place occupee

          tantque (j < k) faire
          begin
           i := i + 1;
           if (etat[i] == 1) then
              j := j + 1;
          end;

          message dem (1, park[i]);  -- envoie de la voiture
          park[i] := -1;             -- vers le controleur
          etat[i] := 0;
          cpt := cpt - 1;
        end;
        else
        begin
          message dem (0, null);
        end;
      end;
      if (rqt == 5) then
      begin
```

```
        fini := 1;
      end;
    end;
    fin;
  end
};

controleur( pere, adr_t_e, adr_t_s, nb_pst)
{

  type := CRO;
  charge_calcul := 5;

  var ok, clef, i, j, rep, bon, le_pst, pst_lav[50], null, reste, q;
  begin
    null := -1;

    i := 0;
    tantque (i < nb_pst) faire
    begin
      pst_lav[i] := null;
      i := i + 1;
    end;

    tantque (etat_global == 0) faire -- boucle principale
    begin

       -- recherche d'un poste de lavage libre
      i := 0;
      tantque ((pst_lav[i] != null) && (i < nb_pst)) faire
        i := i + 1;

      if ( i == nb_pst) then  -- tous les postes sont occupes
      begin
        attente (clef);  -- attente de la liberation d'un poste
        i := clef;
      end;

      q := cpt_ch / 4;
      reste := cpt_ch - (q * 4);
      cpt_ch := cpt_ch + 1;

      if (reste == 0) then
        pst_lav[i] := creation poste_lavage(self, i, adr_t_s);
      if (reste == 1) then
```

```
        pst_lav[i] := creation poste_lavage1(self, i, adr_t_s);
      if (reste == 2) then
        pst_lav[i] := creation poste_lavage2(self, i, adr_t_s);
      if (reste == 3) then
        pst_lav[i] := creation poste_lavage3(self, i, adr_t_s);


      bon := 0;

       -- recherche d'une voiture dans le troncon

      tantque ((bon == 0) && (etat_global == 0)) faire
      begin

        message adr_t_e (self, 3); -- demande au troncon de la station
        attente [adr_t_e] (ok, clef); -- une voiture a laver
        if (ok != 0) then -- il y a une voiture a laver
        begin
          bon := 1;
        end;
        calcul [20];
      end;

      if (etat_global == 1) then
      begin
        message pst_lav[i] (self, null);
      end;

      if (bon == 1) then
      begin
-- fournir a la station de lavage
        message pst_lav[i] (self, clef);-- j'envoie la voiture vers le
-- poste de lavage
      end;

    end;

    i := 0;
    j := 0;
    tantque (i < nb_pst) faire
    begin
      if (pst_lav[i] != null) then
        j := j + 1;
      i := i + 1;
    end;
```

```
    i := 0;
    tantque (i < j) faire
    begin
      attente (clef);
      i := i + 1;
    end;
    message pere (self);
    fin;
  end
};

voiture(pere, adr_t, num)
{
  type := CRO;
  charge_calcul := 2;

  var i, rep, next, fini, the_next, crt_t, sortie_lavage, lavage;

  begin
    fini := 0;
    sortie_lavage := 0;
    crt_t := -1;
    next := adr_t;
    tantque (fini == 0) faire
    begin
      tantque (next != -1) faire
      begin
i := (num * 2) + 1;
        calcul [i];
        message next (self, 1); -- Peut - on avancer vers le troncon
        attente [next] (rep, the_next); -- suivant

        if (rep == 1) then -- oui troncon classique
        begin
          if (crt_t != -1) then
            message crt_t (self, 2); -- sortie du troncon courant
          crt_t := next; -- entree dans le troncon suivant
          next := the_next;
          if (sortie_lavage == 1) then
          begin
            message lavage (1);
            sortie_lavage := 2;
          end;
        end;
        if (rep == 2) then -- entree dans la station de lavage
```

```
        begin
          if (crt_t != -1) then
            message crt_t (self, 2); -- sortie du troncon courant
-- entree dans le troncon suivant
          attente (lavage, next); -- prise en charge par le controleur
          sortie_lavage := 1;
          crt_t := -1;
        end;

        calcul [50];

      end;

      message pere (self,num);
      fini := 1;
    end;
    fin;
  end
};

poste_lavage(pere, num, adr_t)
{

  type := CEM;
  charge_calcul := 1;

 var dem, fini, rqt, etat, clef, next, ok, bon, wker;
  begin
    attente [pere] (dem, clef); -- reception de la voiture
    if (clef != -1) then
    begin
      calcul [10]; -- mouillage
      calcul [50]; -- lavage
      calcul [10]; -- rincage
      calcul [20]; -- sechage
      message clef (self, adr_t);
      attente [clef] (ok);
    end;
    message pere (num);
    fin;
  end
};

poste_lavage1(pere, num, adr_t)
{
```

```
  type := CEM;
  charge_calcul := 2;

 var dem, fini, rqt, etat, clef, next, ok, bon, wker;
  begin
    attente [pere] (dem, clef); -- reception de la voiture
    if (clef != -1) then
    begin
      calcul [150]; -- entretien
      calcul [10]; -- mouillage
      calcul [50]; -- lavage
      calcul [10]; -- rincage
      calcul [20]; -- sechage
      message clef (self, adr_t);
      attente [clef] (ok);
    end;
    message pere (num);
    fin;
  end
};

poste_lavage2(pere, num, adr_t)
{

  type := CEM;
  charge_calcul := 2;

 var dem, fini, rqt, etat, clef, next, ok, bon, wker;
  begin
    attente [pere] (dem, clef); -- reception de la voiture
    if (clef != -1) then
    begin
      calcul [250]; -- preparation
      calcul [10]; -- mouillage
      calcul [80]; -- lavage
      calcul [250]; -- finition
      message clef (self, adr_t);
      attente [clef] (ok);
    end;
    message pere (num);
    fin;
  end
};
```

```
poste_lavage3(pere, num, adr_t)
{

  type := CEM;
  charge_calcul := 1;

 var dem, fini, rqt, etat, clef, next, ok, bon, wker;
  begin
    attente [pere] (dem, clef); -- reception de la voiture
    if (clef != -1) then
    begin
      calcul [10]; -- rincage
      calcul [20]; -- sechage
      message clef (self, adr_t);
      attente [clef] (ok);
    end;
    message pere (num);
    fin;
  end
};
```

Annexe C

Exemples d'applications pour le run-time CAC

Chaque application se compose d'un ensemble de comportements qui se présentent comme une fonction C. Ces fonctions sont ajoutées dans le code des modules sous la forme de fichier *include*.

C.1 Factorielle

C.1.1 Le comportement *fac*

```
void fac(Component my, int *arg)
{
 Mess_Ptr mes1,mes2;
 char *para;
 int borne_inf, borne_sup, result;
 Component sender, fils_g, fils_d;
 long i, max;
 float a,b;

  sender = arg[0];
  borne_inf = arg[1];
  borne_sup = arg[2];
  b = 1;
```

```
  (void)InitData(my,0,B_FAC,0,NULL);

  if (borne_inf == borne_sup) {
    SendReply(my,sender,sizeof(int),(char *) &(borne_inf));
  } else {
    para = (char *)ArgToArray(3,my,borne_inf,(borne_inf+borne_sup)/2);
    fils_g =NewComponent(my,B_FAC1,CEM,3*sizeof(int),para);
    dispose(para);
    para = (char *)ArgToArray(3,my,
                    ((borne_inf+borne_sup)/2)+1,borne_sup);
    fils_d =NewComponent(my,B_FAC,CEM,3*sizeof(int),para);
    dispose(para);
    GetMessage(my,my,&mes1);
    GetMessage(my,my,&mes2);
    result = *ArgMess(mes1,1) + *ArgMess(mes2,1);
    SendReply(my,sender,sizeof(int),(char *) &(result));
    FreeMessage(mes1);
    FreeMessage(mes2);
  }
  EndComponent(my);
}
```

C.1.2 Le module qui utilise *fac*

```
/*****************************************************************/
/* */
/* Module_Dia.c */
/* */
/*****************************************************************/

#include <Behavior.h>
#include<Prim_Cac.h>

#include<fac.h>
#include<fac1.h>
#include<fac2.h>

/* #include <constantes.h> */

void declenche_fac(Component my, int n)
{
 Mess_Ptr mes;
 char *para;
```

```
 Component C_fac;
 int result, borne_inf, borne_sup;

    borne_inf = 1;
    borne_sup = n;
    para = (char *)ArgToArray(3,my,borne_inf,borne_sup);
    C_fac=NewComponent(my,B_FAC,CEM,3*sizeof(int),para);
    dispose(para);
    GetMessageTF(my,my,M_USER, M_REPLY,&mes);
    result = *(int *)GetFirstArg(mes, sizeof(int));
    PrintString(my,"Resultat de la somme de 1 a ");
    PrintInt(my,n);
    PrintString(my," = ");
    PrintInt(my,result);
    PrintString(my,"\n");
    Flush(my);
}

void lance(Component my, int *arg)
{
 Mess_Ptr mes;
 char *para;
 int i, j, TIME, tps;
 Component cac_fichier;

   (void)InitData(my,0,B_DIA,0,NULL);

  PrintString(my,"\n*****APPLI*****    \n\n"); Flush(my);

  cac_fichier = OpenFlot(my,WRITE_ONLY,"resultat.tmp");


  TIME = _cputime();

  declenche_fac(my,200);


  PrintString(my,"\n");
  PrintString(my,"temps execution ");
  tps = _cputime() - TIME;
  PrintInt(my, tps);
  PrintString(my," 1/100 s \n");
  Flush(my);
  FPrintInt(my,cac_fichier,tps);
  FFlush(my,cac_fichier);
```

```
  CloseFlot(my,cac_fichier);
  StopModules(my);
  EndComponent(my);
}


int main(int argc, char ** argv)
{
 word create;
  InitModule(argc,argv,6);
  InsertBehavior(lance,B_DIA,1);
  InsertBehavior(cac_flot,B_FLOT,1);
  InsertBehavior(fac,B_FAC,1);
  InsertBehavior(fac1,B_FAC1,10);
  InsertBehavior(fac2,B_FAC2,100);
  DiffuseName();
  create = NewBox(0);
  NewComponent(create,B_DIA,VOID,0,NULL);
  EndModule();
}
```

Bibliographie

[Agh86] Agha (G.). – *Actors. A Model of Concurrent Computation in Distributed Systems.* – The MIT Press, 1986.

[AH91] America (P.) et Hulshof (B.). – *Definition of POOL2/PTC, a parallel object-oriented language.* – Rapport technique, Philips Research Laboratories, Eindhoven - University of Amsterdam, mars 1991.

[All80] Allen (A.O.). – Queueing Models of Computer Systems. *IEEE Computer*, avril 1980, pp. 13–24.

[Ame87] America (P.). – POOL-T : A Parallel Object-Oriented Language. *In : Object-Oriented Concurrent System*, pp. 199–220.

[AP88] André (F.) et Pazat (J.-L.). – Le placement de tâches sur des architectures parallèles. *Technique et Science Informatiques*, vol. 7, n° 4, 1988.

[AS88] Athas (W. C.) et Seitz (C. L.). – Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, août 1988, pp. 9–24.

[Ath87] Athas (W. C.). – *Fine Grain Concurrent Computations, 5242:TR:87.* – Thèse, California Institute of Technology Pasadena, California, mai 1987.

[BAAD91] Babaoglu (O.), Alvisi (L.), Amoroso (A.) et Davoli (R.). – Mapping parallel computations onto distributed systems in paralex. *In : Proc. IEEE CompEuro'91 Conference*, pp. 123–130.

[BB91] Banâtre (J.-P.) et Banâtre (M.). – *Les systèmes distribués - Expérience du projet GOTHIC.* – InterEditions, 1991.

[BCS89] Billionnet (A.), Costa (M.-C.) et Sutter (A.). – Les probl mes de placement dans les syst mes distribu s. *Technique et Science Informatiques*, vol. 8, n° 4, 1989.

[BDG+93] Beguelin (A.), Dongarra (J. J.), Geist (G. A.), Jiang (W.), Manchek (R.), Moore (K.) et Sunderam (V. S.). – *The PVM Project.* – Ornl/tm-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, mai 1993.

[Ben87] Bennet (J.K.). – The design and implementation of Distributed Smalltalk. *In : OOPSLA'87 Proc. of the second ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, Orlando, Florida. Special Issue of SIGPLAN Notices*, pp. 318–330.

[BKT90] Bal (H. E.), Kaashoek (M. F.) et Tanenbaum (A. S.). – Experience with distributed programming in *orca. In : Int'l Conf. on Comp. Languages'90, IEEE.*

[Bla90] Blank (Tom). – The MasPar MP-1 architecture. *In : Proceedings of the IEEE Compcon Spring 1990*, pp. 20–24. – San Francisco, CA, février 1990.

[Bla92] Blake (B. A.). – Assignment of Independent Tasks to Minimize Completion Time. *Software — Practice and Experience*, vol. 22, n° 9, septembre 1992, pp. 723–734.

[BM88] Bollinger (S. W.) et Midkiff (S. F.). – Processor and link assignment in multicomputers using simulated annealing. *In : Int. Conf. on Parallel Processing '88*, pp. 1–7.

[Bok81a] Bokhari (S.H.). – On the Mapping Problem. *IEEE Transactions on Computers*, vol. C-30, n° 3, mars 1981, pp. 207–214.

[Bok81b] Bokhari (S.H.). – A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System. *IEEE Transactions on Software Engeneering*, vol. SE-7, n° 6, novembre 1981, pp. 583–589.

[BS85] Barak (A.) et Shiloh (A.). – A Distributed Load-balancing Policy for a Multicomputer. *Software — Practice and Experience*, vol. 15, n° 9, septembre 1985, pp. 901–913.

[BSS91] Bernard (G.), Stève (D.) et Simatic (M.). – Placement et migration de processus dans les systèmes répartis faiblement couplés. *Technique et Science Informatiques*, vol. 10, n° 5, 1991.

[BT83] Bannister (J. A.) et Trivedi (K.S.). – Task Allocation in Fault-Tolerant Distributed Systems. *Acta Informatica*, vol. 20, 1983, pp. 261–281.

[BYA89] Bhuyan (L. N.), Yang (Q.) et Agrawal (D. P.). – Performance of Multiprocessor Interconnection Networks. *IEEE Computer*, février 1989, pp. 25–37.

[CA82] Chou (T.C.K.) et Abraham (J. A.). – Load Balancing in Distributed Systems. *IEEE Transactions on Software Engeneering*, vol. SE-8, n° 4, juillet 1982, pp. 401–412.

[CC88] Carlier (J.) et Chrétienne (P.). – *Problèmes d'ordonnancement : Modélisation, Complexité, Algorithmes.* – Masson Ed., 1988.

[CHLE80] Chu (W.W.), Holloway (L.J.), Lan (M.-T.) et Efe (K.). – Task Allocation in Distributed Data Processing. *IEEE Computer*, vol. 13, 1980, pp. 57–69.

[Cho90] Chowdhury (S.). – The Greedy Load Sharing Algorithm. *Journal of Parallel and Distributed Computing*, no9, 1990, pp. 93–99.

[CK79] Chow (Y.-C.) et Kolher (W. H.). – Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System. *IEEE Transactions on Computers*, vol. C-28, n° 5, mai 1979.

[CK88] Casavant (T. L.) et Kuhl (J. G.). – A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engeneering*, vol. 14, n° 2, février 1988, pp. 141–154.

[Col91] Colin (J.-Y.). – *Problèmes d'ordonnancement avec delais de communication : Complexité et Algorithmes.* – Thèse, Université Paris VI, février 1991.

[Cou92] Courtrai (L.). – *Les Composants Actifs de Communication: Outils pour la conception et l'implantation de langages parall les objets actifs pour machines MIMD.* – Thèse, Université des Sciences et Technologies de Lille, octobre 1992.

[CRGM92] Courtrai (Luc), Roos (Jean-Francois), Geib (Jean-Marc) et Méhaut (Jean-Francois). – *Les Composants Actifs de Communication: Manuel*

de Programmation (V 2.0). – Rapport technique n° ERA-115, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, Septembre 1992.

[CS93] Cosnard (M.) et Sansonnet (J.-P.). – Machines Très Parallèles. *Le courrier du CNRS dossiers scientifiques*, no80, février 1993.

[CT93] Cosnard (M.) et Trystram (D.). – *Algorithmes et architectures parallèles.* – InterEditions, 1993.

[Das92] Dasnulmadi (O. N. P.). – Les algorithmes de la répartition dynamique de la charge sur une architecture parallèle à couche reconfigurable. *In : 2ème Journées ArMen, PSD-ENST Bretagne*, pp. 30–39.

[DFG90] Duboux (T.), Ferreira (A.) et Gastaldo (M.). – Machine dictionnaire sur architectures à mémoire distribuée. *In : 5èmes rencontres sur le parallélisme*, pp. 213–216. – Brest, mai 1990.

[Dum92] Dumoulin (Cedric). – *Un Ramasse-miettes pour les Composants Actifs de Communication.* – Rapport technique, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, September 1992.

[ELZ86] Eager (D.L.), Lazowska (E.D.) et Zahorjan (J.). – Adaptative Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engeneering*, vol. SE-12, n° 5, mai 1986, pp. 662–675.

[Fol93] Folliot (B.). – *Méthodes et outils de partage de charge pour la conception et la mise en oeuvre d'applications dans les systèmes répartis hétérogènes.* – Thèse, Université de Paris VI, avril 1993.

[FP89] Fdida (S.) et Pujolle (G.). – *Modèles de systèmes et de réseaux, Tome 1.* – Eyrolles, 1989.

[FYN88] Ferguson (D.), Yemini (Y.) et Nikolaou (C.). – Microeconomic Algorithms for Load Balancing in Distributed Computer Systems. *In : Proc. of 8th Int. Conf. in Distributed Computer Systems, San Jose, California.*

[FZ86] Ferrari (D.) et Zhou (S.). – A Load for Index Dynamic Load Balancing. *In : Proc. 1986 Fall Joint Computer Conference, Dallas, Texas*, pp. 684–690.

[FZ87] Ferrari (D.) et Zhou (S.). – An Empirical Investigation of Load Indices for Load Balancing Applications. *In : Proc. Performance'87, 12th IFIP WG7.3 Int. Symposium on Computer Performance, Brussels.*

[GC92] Geib (J.-M.) et Courtrai (L.). – Abstractions for synchronization to inherit synchronization contraints. *In : ECOOP'92 Workshop on Object-Based Concurrency and reuse, Utrecht, the Nederlands.*

[GGG93] Geib (J.-M.), Gransart (C.) et Grenot (C.). – Distributed Object in BOX. *In : TOOLS'93, Workshop on Distributed Objects and Concurrency, Versailles.*

[GLR84] Gao (A.C.), Liu (J.W.S.) et Railey (M.). – Loadbalancing algorithms in homogeneous distributed systems. *In : IEEE Int, Conf. on Parallel Proc.*, pp. 302–306.

[Gmb90] GmbH (Parsytec). – *Multicluster-2. Technical Documentation. Installation, expansion and maintenance manual Rev. 1.1.* – Rapport technique, Juelicher straße 338, D-5100 Aachen, Parsytec GmbH, mai 1990.

[HE91] Heath (M. T.) et Etheridge (J. A.). – Visualizing the performance of parallel programs. *IEEE Software*, vol. 8, n° 5, septembre 1991, pp. 29–39.

[HLDM91] Hemery (Fred), Lazure (Dominique), Delattre (Eric) et Méhaut (Jean-Francois). – An Analysis of Communication and Multiprogramming in the Helios Operating System. *Microprocessing and Microprogramming*, vol. 32, n° 1-5, août 1991, pp. 137–144.

[Hoa78] Hoare (C.A.R). – Communicating sequential processes. *Communications of the ACM*, vol. Volume 21, n° 8, août 1978.

[Hsu93] Hsu (T.). – *Proposition d'une architecture de réseau d'interconnexion à reconfiguration dynamique et asynchrone.* – Thèse, Université des Sciences et Technologies de Lille, février 1993.

[IC82] Irani (K.B.) et Chen (K.-W.). – Minimization of Interprocessor Communication for Parallel Computation. *IEEE Transactions on Computers*, vol. 31, n° 11, novembre 1982, pp. 1067–1075.

[JLHB88] Jul (E.), Levy (H.), Hutschinson (N.) et Black (A.). – Fine-Grained Mobility in the Emerald System. *IEEE Transactions on Computers*, vol. Volume 6, n° 6, février 1988, pp. 109–133.

[JV89] Joosen (W.) et Verbaeten (P.). – On the Use of Process Migration in Distributed Systems. *Microprocessing and Microprogramming*, 1989, pp. 49–52.

[KK92] Karypis (G.) et Kumar (V.). – *Unstructured Tree Search on SIMD Parallel Computers.* – Tr 92-21, University of Minnesota, 1992.

[KL87] Krueger (P.) et Livny (M.). – The Diverse Objectives of Distributed Scheduling. *In : Proc. of 7th Int. Conf. on Distributed Computing Systems, Berlin.*

[KL88a] Kruatrachue (B.) et Lewis (T.). – Grain Size Determination for Parallel Processing. *IEEE Software*, vol. 5, n° 1, janvier 1988, pp. 23–32.

[KL88b] Krueger (P.) et Livny (M.). – A Comparison of Preemtive and Non-Preemptive Load Distributing. *In : Proc. of 8th Int. Conf. on Distributed Computing Systems, San Jose, California.*

[KM87] Kramer (O.) et Muhlenbein (H.). – Mapping strategies in message based multiprocessor systems. *In : PARLE'87 Conf., LNCS*, éd. par Verlag (Springer), pp. 213–225.

[KP92] Kitajima (J. P.) et Plateau (B.). – Building Synthetic Parallel Programs: The Projet *alpes. In : IFIP Transactions, Programming Environments for Parallel Computing*, pp. 161–171.

[KR91] Kumar (V.) et Rao (V.). – Scalability of parallel algorithms for the all-pairs shortest path problem: A summary of results. *Journal of Parallel and Distributed Computing*, vol. 13, 1991, pp. 124–138.

[KTP93] Kitajima (João Paulo), Tron (Cécile) et Plateau (Brigitte). – ALPES: a tool for the performance evaluation of parallel programs. *In : Environments and Tools for Parallel Scientific Computing*, éd. par Dongarra (J. J.) et Tourancheau (B.). pp. 213–228. – Amsterdam, The Netherlands, 1993.

[KW90] Kafura (D. G.) et Washabough (D.). – Garbage Collection of Actors. *In : ECOOP/OOPSLA'90, Ottawa, Canada.*

[KW91] Kuchen (H.) et Wagener (A.). – Comparaison of Dynamic Load Balancing Strategies. *In : Parallel and Distributed Processing*, éd. par Boyanov (K.). pp. 303–314. – Elsevier Science Publishers B. V. (North-Holland).

[LA87] Lee (S.-Y.) et Aggarwal (J. K.). – A Mapping Strategy for Parallel Processing. *IEEE Transactions on Computers*, vol. C-36, n° 4, avril 1987, pp. 433–442.

[Law75] Lawrie (D.H.). – Acces and alignment of data in a array processor. *IEEE Transactions on Computers*, vol. 24, n° 12, décembre 1975, pp. 173–183.

[Len93] Lenstra (J. K.). – Approximation algorithms for job shop scheduling. *In : Workshop on Models and Algorithms for Planning and Scheduling Problems.*

[Lit92] Litzler (L.). – *Contributions à l'étude du paradigme acteur implantation sur Transputer expression des algorithmes distribués.* – Thèse, Université de Franche-Comté, novembre 1992.

[LK87] Lin (F. C. H.) et Keller (R. M.). – The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engeneering*, vol. SE-13, n° 1, janvier 1987, pp. 32–38.

[Lo84] Lo (V. M.). – Heuristic Algorithms for Task Assignment in Distributed Systems. *In : 4th Int. Conf. on Distributed Computing systems*, pp. 30–39.

[LSS92] Luque (E.), Suppi (R.) et Sorribes (J.). – Designing parallel systems: a performance prediction problem. *Microprocessors and Microsystems*, vol. 16, n° 1, 1992, pp. 25–36.

[Ma84] Ma (R. P.). – A model to solve timing-critical application problems in distributed computer systems. *IEEE Computer*, vol. 18, n° 1, janvier 1984.

[Mar92] Marquet (P.). – *Langages explicites à parallélisme de données.* – Thèse, Université des Sciences et Technologies de Lille, février 1992.

[Mey88] Meyer (B.). – *Object-oriented Software Construction.* – Prentice-Hall, 1988.

[MFL93] Murer (S.), Feldman (J. A.) et Lim (C.-C.). – *pSather: Layered Extentions to an Object-Oriented Language for Efficient Parallel Computation.* – Rapport technique n° TR-93-028, ICSI and Computer Science Division, U.C. Berkeley, ICSI and Computer Science Division, U.C. Berkeley, juin 1993.

[MLT82] Ma (P.-Y. R.), Lee (E. Y. S.) et Tsuchiya (M.). – A Task Allocation Model for Distributed Computing Systems. *IEEE Transactions on Computers*, vol. C-31, n° 1, janvier 1982, pp. 41–47.

[MT91] Muntean (T.) et Talbi (E.-G.). – Méthodes de placement statique des processus sur architectures parallèles. *Technique et Science Informatiques*, vol. 10, n° 5, 1991, pp. 355–373.

[Nic91] Nicolas (J.-C.). – *Machines bases de données parallèles: Contribution aux problèmes de la fragmentation et de la distribution.* – Thèse, Université des Sciences et Technologies de Lille, janvier 1991.

[Noa92] Noailles (Frédéric). – *Evaluation des performances d'un interpréte du langage d'acteurs PLASMA-II après portage sur un réseau de transputers et sur une machine multiprocesseurs.* – Rapport technique, Institut de Recherche en Informatique de Toulouse, Institut de Recherche en Informatique de Toulouse, Juin 1992.

[Paz89] Pazat (J.-L.). – *Outils pour la programmation d'un multiprocesseur à mémoires distribuées.* – Thèse, Université Bordeaux I, février 1989.

[Paz93] Pazat (J. L.). – Langages. *In : Ecole d'automne CAPA 93, Port d'albret, Landes*, pp. 55–66.

[Per89] Perihelion Software (édité par). – *The HELIOS Operating System.* – Prentice-Hall, 1989.

[PFK93] Powley (C.), Ferguson (B.) et Korf (R. E.). – Depth-first heuristic search on a simd machine. *Artificial Intelligence*, vol. 60, avril 1993, pp. 199–242.

[PTS88] Pulidas (S.), Towsley (D.) et Stankovic (J. A.). – Imbedding Gradient Estimators in Load Balancing Algorithms. *In : Proc. 8th Int. Conf. on Distributed Computing Systems, San Jose, California*, pp. 482–490.

[RCM93] Roos (Jean-Francois), Courtrai (Luc) et Méhaut (Jean-Francois). – Execution Replay of Parallel Programs. *In : Proceedings of the Euromicro Workshop on Parallel and Distributed Processing.* pp. 429–434. – IEEE.

[Roo94] Roos (J.-F.). – *Mise au point d'applications distribuées pour environnement de développement basé sur une technologie objet.* – Thèse, Université des Sciences et Technologies de Lille, février 1994.

[Rou93] Roucairol (C.). – Algorithme combinatoire parallèle. *In : Ecole d'automne CAPA 93, Port d'albret, Landes*, pp. 207–210.

[RVV92] Roch (J.-L.), Vermeerbergen (A.) et Villard (G.). – Cost prediction for load-balancing: application to Algebraic Computations. *In : Parallel Processing: CONPAR92 - VAPP V, Lyon France*, pp. 467–478.

[RZZ93] Rudich (A.), Zernik (D.) et Zodik (G.). – Visage - visualisation of attribute graphs: A foundation for a parallel programming environment. *In : Environments and Tools for Parallel Scientific Computing*, pp. 171–192.

[SB88] Schelvis (M.) et Bledoeg (E.). – The Implementation of a Distributed Smalltalk. *In : ECOOP'88, Oslo, Norway, Lectures Notes in Computer Science 322*, éd. par Springer-Verlag, pp. 212–232.

[SCM+91] Shapiro (M.), Courhant (Y.), Marzul (J.-P. Le), Makpangou (M.), Ruffin (M.) et Valot (C.). – Un bilan du système réparti à objet sos. *In : AFCET/INTERFACE Numéro 103/104*, pp. 46–53.

[Sim88] Simulog (édité par). – *Manuel de référence du QNAP2.* – Simulog, 1988.

[SK90] Shivaratri (N. G.) et Krueger (P.). – Two Adaptative Location Policies for Global Scheduling Algorithms. *In : Proc. 10th Int. Conf. on Distributed Computing Systems, Paris*, pp. 502–509.

[SS84] Stankovic (J. A.) et Sidhu (I. S.). – An Adaptative Bidding Algorithms for Processes, Clusters, and Distributed Groups. *In : Proc. of Int. Conf. in Distributed Computer Systems, IEEE*, pp. 49–59.

[ST85] Shen (C.-C.) et Tsai (W.-H.). – A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion. *IEEE Transactions on Computers*, vol. C-34, n° 3, mars 1985, pp. 197–203.

[Sta85] Stankovic (J.A.). – Stability and distributed scheduling algorithms. *IEEE Transactions on Software Engeneering*, vol. 11, n° 10, octobre 1985, pp. 1141–1152.

[Sto77] Stone (H. S.). – Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engeneering*, vol. SE-3, n° 2, janvier 1977, pp. 85–93.

[Stu88] Stumm (M.). – The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster. *In : 2nd IEEE Conference on Computer Workstations.*

[Sun88] Sun Microsystems (édité par). – *System Services Overview Chapter 6: Lightweight Processes part number 800-1753.* – Sun Microsystems, 1988.

[Tal91] Talbi (E.-G.). – Un algorithme d'allocation dynamique de processus sur un réseau de transputers. *La lettre du TRANSPUTER*, no11, septembre 1991, pp. 7–20.

[TH86] Tripathi (S. K.) et Huang (S.-T.). – Distributed resource scheduling for a large scale network of processors : Hcsn. *In : The 6th Int. Conf.*

on Distributed Computing Systems, Cambridge, Massachusetts, pp. 321–328.

[Thi91] Thinking Machines Corpporation (édité par). – *The Connection Machine CM-5 Technical Summary.* – Thinking Machines Corpporation Cambridge, Massachusetts, 1991.

[TL89] Theimer (M.M.) et Lantz (K. A.). – Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engeneering*, vol. 15, n° 11, novembre 1989, pp. 1444–1458.

[Weg90] Wegner (P.). – Concepts and Paradigms of Object-Oriented Programming. *In : OOPS messenger, A Quarterly Publication of the Special Interest Group on Programming Languages, ACM press, v1*, pp. 8–87.

[Win92] Winckler (A.). – Load Balancing and Execution Control - An Approch to Classification. *In : Proc. of ISMM Conference on Parallel and Distributed Computing and Systems*, pp. 140–146.

[WM85] Wang (Y.) et Morris (R.). – Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, vol. C-34, n° 3, mars 1985.

[WSB90] Whiley (D.), Starkweather (T.) et Bogart (C.). – Genetic algorithms and neural networks: optimizing connections and connectivity. *Parallel Computing*, vol. 14, n° 3, août 1990, pp. 347–361.

[YT87] Yonezawa (A.) et Tokoro (M.). – *Object-Oriented Concurrent Programming.* – The MIT Press, 1987.

[ZF87] Zhou (S.) et Ferrari (D.). – A Measurement Study of Load Balancing Performance. *In : Proc. 7th Int. Conf. on Distributed Computing Systems, Berlin.*

[Zho88] Zhou (S.). – A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Transactions on Software Engeneering*, vol. 14, n° 9, septembre 1988, pp. 1327–1341.