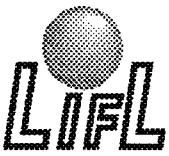
NUMERO D'ORDRE : 1948 ANNÉE : 1997


Laboratoire d'Informatique Fondamentale de Lille




# THÈSE

présentée à

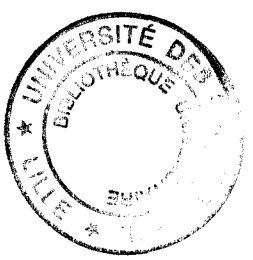
**L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE**

pour obtenir le titre de

**DOCTEUR EN INFORMATIQUE**

par

**Nouredine MELAB**




# Gestion de la granularité et régulation de charge dans le modèle $P^3$ d'évaluation parallèle des langages fonctionnels

**Thèse soutenue le 08 janvier 1997, devant la commission d'examen :**

| | |
|---|---|
| Président : | Jean-Marc GEIB |
| Directeur de thèse : | Bernard TOURSEL |
| Rapporteurs : | Bertil FOLLIOT |
| | Jacques JULLIAND |
| Examinateurs : | Nathalie DEVESA |
| | Marie-Paule LECOUFFE |

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

Mes vifs remerciements vont à Monsieur *Jean-Marc Geib*, Professeur à l'université de Lille I et directeur du LIFL, d'avoir accepté de présider le jury de cette thèse.

Je remercie vivement Monsieur *Bertil Folliot*, Maître de conférences à l'université Pierre et Marie Curie et habilité à diriger des recherches, de m'avoir fait l'honneur de rapporter cette thèse malgré une forte contrainte de temps. Je le remercie aussi pour ses remarques et critiques qui m'ont aidé à voir mieux comment améliorer l'évaluation du travail.

Je remercie également avec intensité Monsieur *Jacques Julliand*, Professeur à l'université de Franche-Comté et directeur du LIB (Laboratoire d'Informatique de Besançon), de m'avoir également fait l'honneur d'être rapporteur de cette thèse. Je le remercie également pour ses remarques et critiques qui m'ont permis d'améliorer ce travail.

Mes remerciements particulièrement chaleureux vont à Monsieur *Bernard Toursel*, Professeur à l'université de Lille I et directeur-adjoint de l'EUDIL, pour avoir accepté de diriger mes travaux. Son sens constructif de la critique et son expérience ont nettement amélioré ma démarche face aux problèmes scientifiques. Par ailleurs, ses rapports conviviaux ont fortifié mon goût pour le travail collectif. Qu'il trouve ici ma profonde gratitude.

Je tiens aussi à remercier vivement Mademoiselle *Nathalie Devesa*, Maître de conférences à l'université de Lille I, d'avoir co-encadré cette thèse. Ses critiques très constructives m'ont été d'un apport scientifique considérable.

Je remercie intensément Madame *Marie-Paule Lecouffe*, Maître de conférences à l'université de Lille I, d'avoir également co-encadré ce travail. Ses conseils et encouragements m'ont beaucoup aidé dans cette étude. Ses critiques et sa lecture rigoureuse de la thèse ont contribué à l'amélioration de la présentation de celle-ci.

Je ne dois pas oublier de remercier Monsieur *El-Ghazali Talbi*, Maître de conférences à l'université de Lille I, pour ses encouragements et toutes les discussions fructueuses que nous avons eues sur la régulation de charge.

Je remercie vivement tous les membres de l'équipe PALOMA pour toute l'aide précieuse qu'ils m'ont apportée. Qu'ils trouvent ici toute ma reconnaissance.

Mes remerciements s'adressent aussi à tous mes collègues du département d'informatique de l'IUT "A" pour avoir aménagé mon emploi du temps de façon à me permettre de finir la rédaction et préparer la soutenance de cette thèse dans de meilleures conditions.

Je remercie Madame *Isabelle Wattier* pour avoir imprimé cette thèse. Je tiens à saisir l'occasion ici pour remercier Monsieur *Henri Glanc*, parti en retraite il y a quelques jours, pour m'avoir imprimé avec un sérieux exemplaire des documents qui ont certainement servi à la réalisation de ce travail.

Je tiens à remercier également tou(te)s mes ami(e)s, en particulier *Vassiliki Ziko* et *Carole Corman*, qui m'ont témoigné leur compréhension, leur soutien et leur sympathie pendant la réalisation de ce travail.

Ma famille m'a énormément soutenu et encouragé des années durant. Qu'elle trouve ici toute ma reconnaissance et mes remerciements les plus chaleureux.

À la mémoire de mon père
À ma mère
À mes sœurs et frères

# Table des matières

# Introduction

Les langages fonctionnels sont de bons candidats pour la programmation d'applications parallèles. Ceci au moyen, d'une part, à leurs propriétés mathématiques qui facilitent leur compréhension, leur analyse et leur utilisation, et, d'autre part, au parallélisme implicite et abondant qu'ils véhiculent. De plus, leur propriété de Church-Rosser, selon laquelle le résultat de l'exécution d'un programme ne dépend pas de l'ordre d'évaluation des expressions fonctionnelles le composant, rend le problème de synchronisation bien moins complexe.

Cette propriété de confluence a permis l'émergence de différents modèles d'évaluation des langages fonctionnels. Ceux-ci sont répartis en deux classes : les *modèles dataflow* et les *modèles de réduction.* Les modèles *dataflow* représentent le programme par un graphe orienté dont les nœuds contiennent des instructions et dont les arcs représentent des canaux logiques à travers lesquels les données arrivent aux instructions qui sont censées les utiliser. Les modèles de réduction, quant à eux, exécutent le programme en effectuant, par application d'un système de règles de réécriture, une série de réductions sur l'expression représentant le programme. Parmi les modèles de réduction existants, on distingue : le modèle de *réduction de chaîne* et le modèle de *réduction de graphe.*

Le premier modèle représente le programme par une chaîne de caractères. L'exécution d'un programme selon ce modèle est, par conséquent, une réécriture de mots. En revanche, dans le deuxième modèle, le programme est un graphe syntaxique dont chaque nœud contient soit une fonction, une donnée ou un nœud application. Pour ce modèle, il s'agit donc, à l'exécution du programme, d'une réécriture de graphe.

Plusieurs implémentations parallèles des langages fonctionnels, basées pour la plupart sur le modèle de réduction de graphe, ont vu le jour, essentiellement pendant les années 80. L'idée initiale derrière celles-ci était de réaliser des machines dédiées à l'exécution des langages fonctionnels telles que ALICE[DR81], ZAPP[BS81], MaRS[CDG+86], Flagship[WSWW87] et GRIP[JCSH87]. Cependant, le coût de réalisation de ces machines et l'absence de portabilité, qui devient de plus en plus une exigence, nuisent à la viabilité d'une telle idée. Aussi, un recours aux implémentations sur des machines conventionnelles est nécessaire. Les machines ALFALFA[Gol88] et GUM[THM+96] sont deux exemples de telles implémentations.

Dans le cadre du projet PARALF[1] de l'équipe PALOMA[2] du LIFL[3], un autre modèle de réduction (parallèle), baptisé $P^3$, a été défini par N. Devesa dans sa thèse [Dev90]. Dans le

[1] **PARA**llélisme et **L**angages **F**onctionnels
[2] **PA**rallélisme **LO**giciel et **MA**tériel
[3] **L**aboratoire d'**I**nformatique **F**ondamentale de **L**ille

même cadre, ce modèle a été ensuite étendu et validé par simulation par N. Bennani dans [Ben94]. A la différence des modèles de réduction évoqués précédemment, le modèle $P^3$ est basé sur une représentation séparée du programme et de sa donnée, ce qui lui confère son originalité. En effet, le programme est représenté, dans un espace appelé *espace d'exploration*, par une forêt d'arborescences fonctionnelles. Les nœuds de ces arborescences contiennent les fonctions du programme. Les données, quant à elles, sont représentées, dans un espace appelé *espace de réduction*, par des arbres de données. Les feuilles de ces arbres contiennent les données du programme. L'évaluation d'un programme, selon $P^3$, utilise deux mécanismes asynchrones : le *mécanisme d'exploration* et le *mécanisme de réduction*. Le premier mécanisme effectue un parcours parallèle des arborescences fonctionnelles et génère en parallèle des ordres de réduction à déstination de l'espace de réduction. Ces ordres sont exécutés en parallèle selon le mécanisme de réduction. Ces trois niveaux de parallélisme ont valu le nom $P^3$ (**P**arallel x **P**arallel x **P**arallel) au modèle.

Le modèle $P^3$ a été simulé dans [Ben94] sur un réseau de processeurs. Les trois activités parallèles du modèle sont mises en œuvre par échange de messages. Les résultats de simulation, obtenus sur quelques exemples de programmes, ont révélé, d'une part, l'importance du parallélisme implicite possible dans le modèle. D'autre part, ils ont mis en évidence un certain nombre de problèmes dont les principaux sont les suivants :

- Pour tous les programmes simulés, les résultats obtenus montrent que le nombre de messages échangés dans et entre les deux espaces est considérable. Par exemple, pour le programme de tri ("Quicksort") de 30 entiers, le nombre de messages générés est d'environ 60000. Par ailleurs, le traitement d'un message consiste quasiment en un envoi d'un autre message. Ce constat met en évidence le caractère fin de la granularité qui est un problème qui compromet, en général, énormement l'efficacité des langages fonctionnels. Aussi, un regroupement de nœuds intra-espace et inter-espaces doit être effectué afin de grossir la granularité des traitements.

- Afin de pouvoir exploiter le parallélisme, les données sont fréquemment, comme le révèlent les résultats de simulation, et dynamiquement copiées. Le mécanisme de copie utilisé doit garantir, d'une part, un bon degré de localité des données et, d'autre part, un bon degré de parallélisme. Pour ce faire, un mécanisme de copie avec seuil de regroupement/éclatement est défini dans [Ben94, BM95]. Par ailleurs, pour assurer une bonne répartition des copies constituées au moyen du seuil, une politique de régulation dynamique de la charge doit être définie.

- L'exécution des fonctions contenues dans les arborescences fonctionnelles nécessite le parcours de celles-ci par échange de messages. Par ailleurs, l'exécution se fait en utilisant une approche interprétée. Ce qui occasionne un surcoût de traitement.

Les trois problèmes ci-dessus constituent l'essentiel de la problématique de cette thèse. Ils y sont abordés avec les objectifs suivants :

- Proposer une approche de gestion de la granularité de parallélisme qui assurera :
  - Le partitionnement du programme en paquets de fonctions ;

  - Le regroupement, dans la mesure du possible, des données de chaque paquet ;
  - L'association des paquets avec leurs données respectives.

- Mettre en place une politique générale (réutilisable) de régulation dynamique de la charge afin de minimiser le temps d'exécution des programmes. Celle-ci devra, d'une part, maintenir un état de charge global dans la machine avec un surcoût de calcul et de communication le moindre possible. D'autre part, elle devra prendre des décisions de transfert de charge et de localisation qui conduiront le moins possible à des inondations de processeurs.

- Implanter le modèle $P^3$ sur une machine MIMD sans mémoire commune en utilisant une approche d'exécution compilée. Une traduction des programmes dans un langage intermédiaire distribué et compilable est alors nécessaire. Par ailleurs, afin de minimiser davantage les communications occasionnées par le modèle, une exécution multithreadée est souhaitable. Dans ce but, l'environnement parallèle et multithreadé $PM^2$ [NM95, Nam97], a été retenu pour l'implantation du modèle.

## Organisation de la thèse

La thèse est organisée en huit chapitres regroupés en trois parties. La première partie définit dans les chapitres 1 et 2 le cadre de la thèse, ses motivations et ses objectifs. La deuxième partie fait un bref état de l'art du problème de granularité, puis décrit l'approche proposée et le modèle d'exécution qui en découle. Cette partie est constituée des chapitres 3, 4 et 5. La troisième partie fait le point sur les méthodes de régulation existantes, puis décrit l'algorithme proposé et enfin présente une évaluation de celui-ci. Cette partie comprend les chapitres 6, 7 et 8. Les conclusions et perspectives du travail, les annexes et un glossaire sont présentés à la fin de la thèse.

Dans le chapitre 1, nous décrivons le langage Graal [Bel86], langage utilisé pour décrire, simuler et implanter le modèle $P^3$. Graal est un langage sans variable qui est une extension du langage FP [Bac78]. Ensuite, nous décrivons succinctement les différents modèles d'évaluation existants des langages fonctionnels, i.e. les modèles *dataflow* et les modèles de réduction, en montrant comment les différentes formes de parallélisme y sont exploitées. Enfin, nous présentons un état de l'art des implémentations parallèles des langages fonctionnels. Nous donnerons d'abord les problèmes posés par celles-ci, puis nous montrons comment ces derniers sont traités dans les machines suivantes : Rediflow [KL84], MaRS [CDG+86], Flagship [Kea94], GRIP [JCSH87] et GUM [THM+96].

Le chapitre 2 présente la définition et la simulation du modèle $P^3$. Dans la partie définition, nous donnons les représentations arborescentes du programme et de la donnée dans les deux espaces du modèle. Ensuite, nous décrivons les deux mécanismes d'exploration et de réduction qui régissent l'exécution d'un programme selon le modèle. Dans la partie simulation, nous décrivons brièvement la méthode de simulation à base de messages utilisée dans la thèse [Ben94]. Nous montrons également les résultats de simulation obtenus sur quelques exemples de programmes. Ces résultats montrent le parallélisme abondant possible dans $P^3$ et mettent en évidence la problématique et les objectifs de la thèse.

Dans le chapitre 3, nous effectuons une classification des méthodes de gestion de la granularité proposées dans la littérature. Suivant l'effort fourni par le programmeur dans la gestion du prob-

# Chapitre 2

# Le modèle $P^3$ : Définition et simulation

## 2.1 Introduction

Le modèle $P^3$ est un modèle d'évaluation parallèle des langages fonctionnels sans variables. Dans ce chapitre, nous allons donner une description succincte de ce modèle. Une description plus élaborée et plus formelle se trouve dans [Ben94].

Le modèle $P^3$ est un modèle de réduction qui a la particularité de représenter le programme et sa donnée séparément. Le programme est représenté, dans un espace appelé *espace d'exploration*, par une forêt d'arborescences fonctionnelles. Les nœuds de ces arborescences contiennent les fonctions du programme.
Les données sont représentées, dans un espace appelé *espace de réduction*, par des arbres de données. Les feuilles de ces arbres contiennent les données du programme.

L'évaluation d'un programme, selon $P^3$, utilise deux mécanismes : le *mécanisme d'exploration* et le *mécanisme de réduction*. Le premier mécanisme fait un parcours parallèle des arborescences fonctionnelles et génère en parallèle des ordres de réduction à destination de l'espace de réduction. Ces ordres sont exécutés en parallèle selon le mécanisme de réduction.

Le présent chapitre est divisé en trois parties. La première partie décrit succinctement le modèle $P^3$ en montrant comment représenter un programme et sa donnée selon ce modèle. Elle décrit également les deux mécanismes d'exploration et de réduction qui contrôlent l'évaluation d'un programme. Le modèle $P^3$ a été simulé dans [Ben94]. La deuxième partie rapporte les choix d'implantation effectués et quelques résultats de simulation. A partir de ces résultats, la troisième partie de ce chapitre dégage la problématique traitée dans la thèse ainsi que les objectifs de celle-ci.

## 2.2 Définition

### 2.2.1 Représentation du programme

Dans les langages fonctionnels sans variables, un programme est constitué d'un ensemble de définitions. L'une, parmi ces dernières, est considérée comme le moteur du programme. Elle est appelée **définition principale** du programme. Par opposition, les autres définitions représentent les **définitions secondaires** du programme. Formellement, une définition est une composition de fonctions où chaque fonction est soit une fonction primitive, une occurrence d'utilisation d'une définition ou une forme fonctionnelle. Par exemple, le programme de la figure 2.1 contient une seule définition.

Chaque définition est représentée dans l'espace d'exploration par une arborescence fonctionnelle. Celle représentant la définition principale est appelée **arborescence fonctionnelle principale.** Une arborescence fonctionnelle est composée d'un ensemble de nœuds reliés par trois types d'arcs : "$\downarrow$", "$\hookrightarrow$" et "$\longrightarrow$". L'arborescence fonctionnelle représentant une définition est constituée de la façon suivante :

- Chaque fonction primitive est représentée par un **nœud fonctionnel** appelé **nœud fonction primitive.**
- Chaque occurrence d'utilisation d'une définition est représentée par un nœud fonctionnel appelé **nœud définition.** Ce nœud référence l'arborescence fonctionnelle représentant la définition utilisée.
- Une forme fonctionnelle admet un ou plusieurs paramètres. Chaque paramètre est une composition de fonctions représentées chacune par une sous arborescence fonctionnelle. La représentation complète d'une forme fonctionnelle est constituée d'un nœud contenant le symbole de la forme fonctionnelle appelé **nœud forme fonctionnelle.** Un arc de type "$\hookrightarrow$" relie ce nœud à la sous arborescence fonctionnelle représentant le premier paramètre. Les arborescences fonctionnelles représentant deux paramètres consécutifs, sont reliées par un arc de type "$\longrightarrow$".
- L'arborescence fonctionnelle représentant la définition d=$f_1 \circ \ldots \circ f_n$ s'obtient en reliant par des arcs de type "$\downarrow$" les représentations des fonctions $f_n$, $f_{n-1}$, $\ldots$, $f_1$.

**Exemple** Le programme qui calcule le produit scalaire contient la définition suivante :

$$def \;\; ps \equiv (binu \;\; apply \;\; +) \;\; \circ \;\; \alpha \;\; *$$

L'arborescence fonctionnelle représentant cette définition est illustrée par la figure 2.1.

**Définitions**

1. **Notion de successeur** : Un nœud extrémité d'un arc de type "$\downarrow$" ayant comme origine un autre nœud fonctionnel $n$, est le **successeur** du nœud $n$.

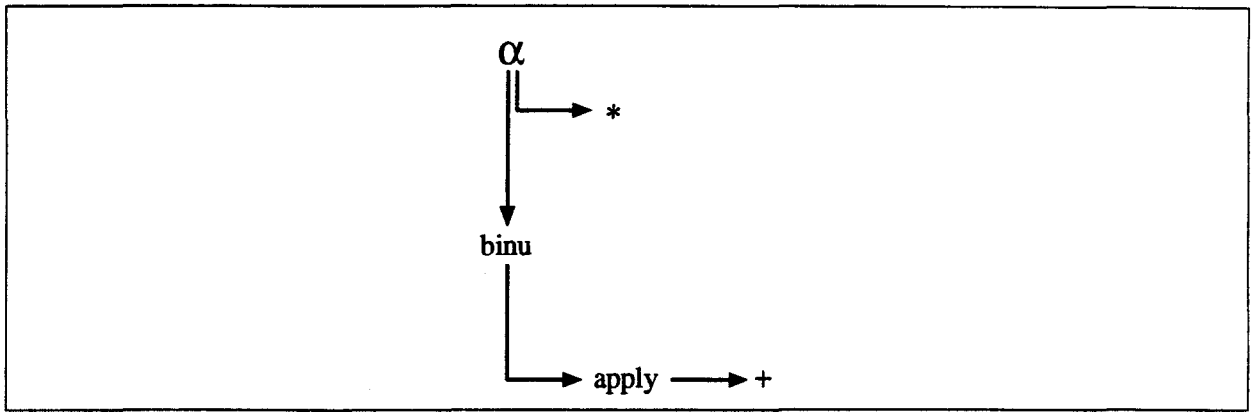



Figure 2.1 : Représentation du produit scalaire dans le modèle $P^3$

2. **Notion de chemin séquentiel** : Les nœuds fonctionnels reliés par des arcs de type "↓" constituent un **chemin séquentiel** de l'arborescence fonctionnelle.

Dans l'exemple illustré par la figure 2.1 :

- Le nœud fonctionnel $\alpha$ est la **racine** de l'arborescence fonctionnelle.
- Les nœuds $\alpha$ et *binu* constituent un chemin séquentiel.
- Le nœud *binu* est successeur du nœud $\alpha$ dans le chemin séquentiel.

### 2.2.2 Représentation de la donnée

Une donnée est soit un atome, une séquence ou une liste. Dans tous les cas, elle est représentée à l'aide d'une structure appelée **arbre de données.** Ce dernier est construit de la façon suivante :

- Un atome est représenté par un **nœud de donnée** contenant la valeur de l'atome.
- Chaque élément d'une séquence ou d'une liste est soit un atome, une séquence ou une liste. Une séquence ou une liste est représentée par un arbre binaire dont les feuilles sont des atomes.

La figure 2.2 schématise l'arbre de données représentant la séquence de listes suivante :

< 0 1 2 > < 3 4 5 >

**Remarque** : En présence de fonctions d'ordre supérieur, certains nœuds de l'espace de réduction sont des nœuds fonctionnels.

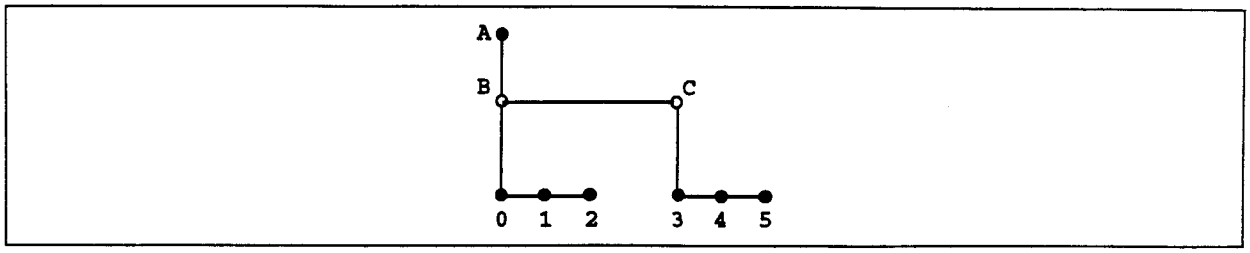



Figure 2.2 : Représentation d'une donnée dans le modèle $P^3$

### 2.2.3 Exécution d'un programme

L'évaluation de l'application d'un programme à son argument dans le modèle $P^3$ se base sur deux activités parallèles : l'**exploration** des arborescences fonctionnelles par activation des nœuds fonctionnels et la **réduction** de l'argument. Dans la suite, nous allons décrire de façon informelle ces mécanismes. Leur description formelle est donnée dans la thèse [Ben94].

#### 2.2.3.1 Mécanisme d'exploration

L'exploration d'une arborescence fonctionnelle a pour but de permettre à chaque nœud fonctionnel d'une arborescence fonctionnelle de "connaître" son argument. Cet argument est désigné par le nœud racine de l'arbre de donnée le représentant. Ce nœud est appelé **nœud cible** du nœud fonctionnel. Activer un nœud fonctionnel revient donc à lui associer son nœud cible.

L'exécution d'un programme, qui se traduit par l'exploration de la forêt d'arborescences fonctionnelles le représentant, s'effectue en respectant les règles suivantes :

- L'exploration commence toujours à la racine de l'arborescence fonctionnelle principale du programme.
- Chaque nœud fonctionnel activé déclenche l'exploration de son successeur, s'il existe et si ce dernier contient une fonction stricte. L'anticipation est possible sur l'exploration des nœuds contenant des fonctions non strictes.
- L'activation d'un nœud définition déclenche l'exploration de l'arborescence fonctionnelle principale de cette définition.
- L'activation d'un nœud forme fonctionnelle déclenche l'exploration parallèle des sous arborescences fonctionnelles représentant ses paramètres.

En résumé, l'exploration d'une arborescence fonctionnelle implique l'exploration d'autres arborescences fonctionnelles en présence de nœuds définitions. Elle provoque également, dans le cas où le programme comporte des formes fonctionnelles, l'exploration parallèle des sous arborescences fonctionnelles de l'arborescence.

### 2.2.3.2 Mécanisme de réduction

Chaque nœud activé effectue une demande de réduction sur l'argument désigné par le nœud cible qu'il a reçu grâce au mécanisme d'exploration. La réduction à effectuer dépend de la fonction représentée par le nœud fonctionnel. Ces réductions sont principalement de deux types : des réductions par des fonctions primitives qui permettent l'obtention d'un résultat, par exemple l'application de la fonction "+" à l'argument < 1 2 3 > permet l'obtention de l'objet résultat "6". Le deuxième type de réduction est engendré par des formes fonctionnelles. Il s'agit plutôt, dans ce cas, de préparer les arguments associés ultérieurement aux paramètres de la forme fonctionnelle à partir de l'argument de celle-ci. Par exemple, une requête de réduction issue d'un nœud fonctionnel représentant la forme fonctionnelle composition "{}" à un argument donné permet la constitution d'un nombre de copies de cet argument égal au nombre de paramètres de la forme fonctionnelle moins un. Le but des copies est de pouvoir appliquer à l'argument les paramètres de la forme fonctionnelle en parallèle.
Il existe un troisième type de réductions, celles provoquées en présence de fonctions d'ordre supérieur. Le traitement de ces réductions se traduit par la construction d'arborescences fonctionnelles.

### 2.2.3.3 Exemple

Examinons brièvement à travers l'exemple du produit scalaire ((*binu aply* +) *o* ($\alpha$ *)) appliqué à la donnée < 0 1 2 > < 3 4 5 > les deux mécanismes précédents. Les différentes étapes de la réduction sont résumées ci-dessous.
1) (*binu apply* +) *o* ($\alpha$ *) : < 0 1 2 > < 3 4 5 >
2) (*binu apply* +) : *:< 0 3 > *:< 1 4 > *:< 2 5 >
3) (*binu apply* +) : 0 4 10
4) *apply* + < 0 4 10 >
5) + : 0 4 10
6) 14

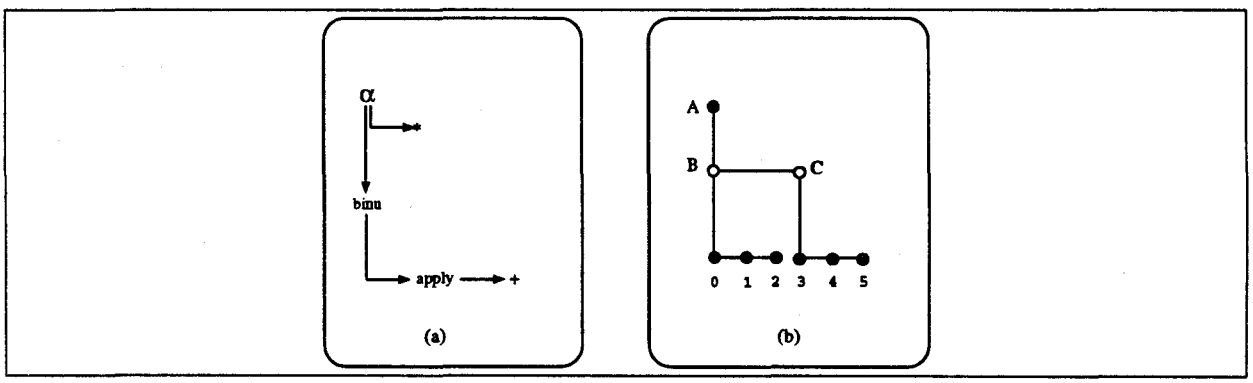



Figure 2.3 : Représentation du produit scalaire dans le modèle $P^3$

Le programme comporte une seule définition. Celle-ci est représentée par une seule arborescence fonctionnelle 2.3. L'exécution du programme se traduit alors par l'exploration de cette dernière qui commence à sa racine i.e. par le nœud fonctionnel A contenant la forme $\alpha$ (distribution généralisée). L'activation de ce nœud (étape 1)) génère un ordre de réduction à destination de son nœud cible i.e. la racine de l'arbre de donnée de la figure 2.3(b). Cet ordre provoque (étape 2)) la distribution des éléments de la première liste à leurs homologues respectifs de la deuxième liste. Il provoque également la distribution de la fonction primitive "*" aux nœuds D, E et F [1]. Cette distribution est illustrée par les flèches en pointillé sur la figure 2.4.(b). L'application de la fonction "*" aux sous-arbres de racines D, E et F se traduit par la réduction de ces derniers. A l'issue des trois réductions (étape 3)), on obtient la séquence 0 4 10 (figure 2.4.(c)). L'activation du nœud "*binu*" (étape 4)) provoque la construction de la séquence d'arguments "+ < 0 4 10 >" puis l'activation du nœud "*apply*". Cette dernière provoque, à son tour, l'exploration du nœud "+" dont l'activation (étape 5)) génère un ordre de réduction à destination de son nœud cible i.e. la racine A de l'arbre de donnée. Cet ordre de réduction déclenche d'autres ordres de réduction comme on peut le voir sur la figure 2.4.(c). Le résultat de la réduction (étape 6)) est un arbre de donnée réduit à un seul nœud contenant la valeur "14".

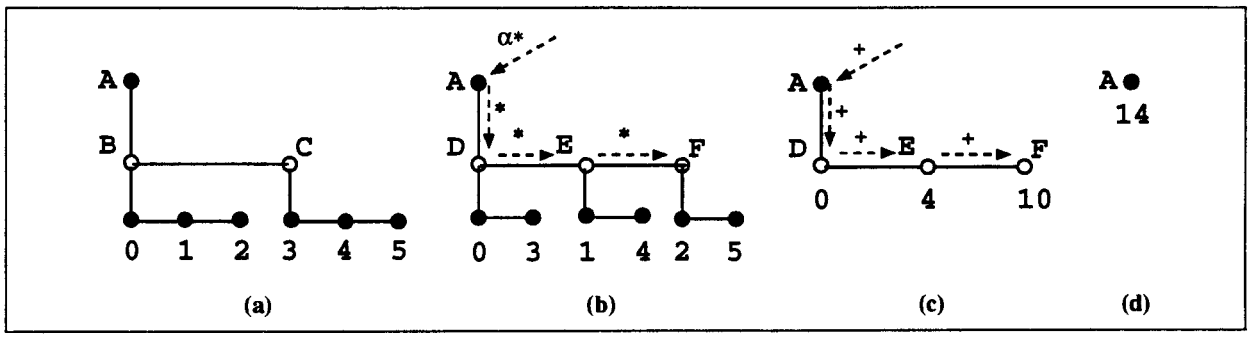



Figure 2.4 : Evolution de la donnée du produit scalaire lors de la réduction

#### 2.2.3.4 Parallélisme

Dans le modèle $P^3$, trois activités s'effectuent en parallèle de manière asynchrone : l'exploration des arborescences fonctionnelles, la construction des redex [2] et l'évaluation des redex. Ces trois niveaux de parallélisme lui ont valu le nom de $P^3$.

Le parallélisme exploité dans $P^3$ est le parallélisme implicite et conservatif. Le parallélisme horizontal est exploité grâce à la présence de formes fonctionnelles dont les paramètres s'appliquent simultanément à leurs arguments respectifs. Dans l'exemple précédent, les réductions des sous arbres de racines D, E et F peuvent être effectuées en parallèle.
Le parallélisme vertical est également exploité en anticipant sur l'exécution des fonctions non

[1] Dans [Ben94], la notion de **fenêtre cible** a été introduite pour désigner la racine A de l'arbre et les nœuds B et C.

[2] Dans $P^3$, un redex (**reducible expression**) est une association entre une expression fonctionnelle et son nœud cible

strictes. Dans l'exemple précédent, l'exploration du nœud fonctionnel *binu* peut être déclenchée avant la fin des réductions de D, E et F.

## 2.3 Simulation

La validation du modèle $P^3$ est passée par deux étapes dans [Ben94]. La première étape est une validation théorique par comparaison de $P^3$ au modèle de réduction de graphe. Le choix de ce dernier se justifie par sa puissance, son efficacité et sa large utilisation. La deuxième étape est une validation par simulation.

La représentation séparée du programme et ses données, qui fait l'originalité du modèle $P^3$, lui confère certains avantages. En effet, comparé au modèle de réduction de graphe, la constitution des redex est moins coûteuse en temps et en espace. Ceci s'explique par le fait que cette constitution est dynamique dans $P^3$. Par contre, dans le modèle de réduction de graphe, les redex sont constitués à l'aide de nœuds application qui font partie de la repésentation du programme. Par ailleurs, l'utilisation de deux espaces dans $P^3$ induit un asynchronisme des opérations d'exploration et de réduction, qui implique un gain de temps d'exécution. La comparaison détaillée entre les deux modèles se trouve dans [BDLT93, Ben94].
La section suivante présente le deuxième type de validation utilisé, en l'occurrence la simulation.

### 2.3.1 Choix d'implantation

#### 2.3.1.1 Simulation du modèle à base de messages

Dans la simulation, les deux mécanismes du modèle $P^3$ sont mis en œuvre par utilisation de messages. Le mécanisme d'exploration utilise les types de messages suivants. Les exemples donnés se rapportent à l'exemple du produit scalaire précédent.

- Les messages d'exploration ou d'activation, échangés entre nœuds fonctionnels. Un message d'activation à destination d'un nœud fonctionnel successeur d'un nœud définition ou d'un nœud forme fonctionnelle est créé bloqué en attente d'un nombre de messages de fin d'exploration afin d'assurer l'ordonnancement.

- Les messages de fin d'exploration à destination de messages d'activation bloqués. Ces messages sont générés à la fin de l'exploration d'une arborescence fonctionnelle secondaire représentant une définition ou à la fin de l'exploration d'une sous arborescence fonctionnelle paramètre d'une forme fonctionnelle.

- Les messages d'acquittement qui sont utilisés pour débloquer l'exploration d'une nouvelle branche séquentielle[3] bloquée afin d'assurer l'ordonnancement. L'algorithme d'ordonnancement est donné dans [Ben94].

Le mécanisme de réduction utilise principalement deux types messages :

[3] tronçon d'un chemin séquentiel

- Les messages NENR (Noeud Exploré vers Noeud à Réduire), qui traduisent les ordres de réduction émis à destination des nœuds cibles de l'espace de réduction. Un exemple de ce type de messages est celui généré par le nœud contenant la fonction $\alpha$ à destination de son nœud cible et contenant la fonction "*".
- Les messages NRNR (Noeud à Réduire vers Noeud à Réduire), utilisés pour mettre en œuvre le processus de réduction. Ce type de messages circule entre les nœuds de la liste $< 0\ 1\ 2 >$ et ceux de la liste $< 3\ 4\ 5 >$ de l'arbre de la figure 2.3(b) lors de l'application de la fonction "+".

Dans le paragraphe suivant, nous illustrons à travers un exemple l'utilisation des différents types de messages lors de l'exécution d'un programme et l'asynchronisme des activités d'exploration et de réduction et la nécessité de leur synchronisation.

#### 2.3.1.2 Exemple

Considérons l'application suivante : $\{+\ car\ (car\ o\ cdr)\} : < 0\ 1\ 2 >$. Cette application est illustrée par la figure 2.5. L'exécution de cette application commence par l'envoi d'un message d'activation à destination du nœud fonctionnel "{}". Le traitement de ce message déclenche deux activités. La première activité est la duplication de l'argument $< 0\ 1\ 2 >$ pour pouvoir appliquer les deux paramètres de la forme "{}" (figure 2.6.(b)). La deuxième activité consiste en l'envoi d'un message d'activation bloqué à destination du nœud "+" et d'un message d'activation à destination de chacun des nœuds racines des deux sous-arborescences fonctionnelles paramètres de la forme "{}". Le traitement de ces derniers messages, à leur tour, déclenche l'exploration des sous-arborescences auxquelles ils sont destinés.

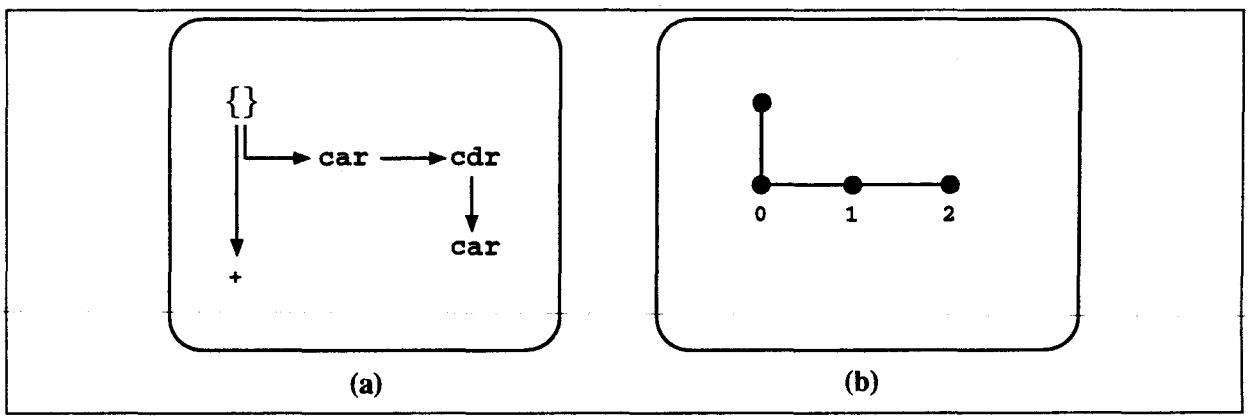



Figure 2.5 : Représentation de l'application $\{+\ car\ (car\ o\ cdr)\} : < 0\ 1\ 2 >$ selon $P^3$

Le premier paramètre est réduit à une seule fonction primitive. Son exploration se traduit par l'envoi d'un message NENR, contenant la fonction "car", à son nœud cible (racine de la première liste) (figure 2.6.(c)). L'exécution du deuxième paramètre est une exploration du chemin séquentiel *car o cdr*. Cette exploration commence par l'activation du nœud *cdr* qui,

à son tour, déclenche l'activation du nœud *car*. La prise en compte de ces messages implique l'envoi de deux messages NENR à destination d'un **même nœud cible** i.e. la racine de la deuxième liste de l'argument (figures 2.6.(b) et 2.6.(c)). Un mécanisme d'ordonnancement est donc nécessaire pour garantir l'exécution de la fonction "cdr" avant la fonction "car". Pour ce faire, un algorithme d'ordonnancement est défini dans [Dev90] et a été légèrement modifié dans [Ben94].

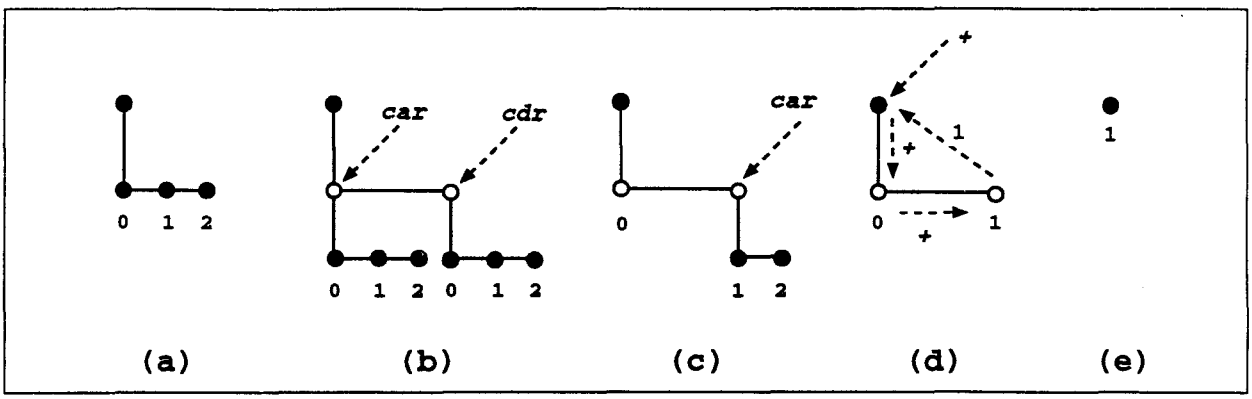



Figure 2.6 : Evolution de la donnée lors de la réduction

Le traitement des messages NENR issus de l'exploration des paramètres met en jeu des messages de réduction de type NRNR. Par exemple, le nœud cible du premier paramètre est réécrit par le contenu du premier élément de la première liste par échange de deux messages NRNR entre ces deux nœuds. Pendant que la réduction s'effectue dans l'espace de réduction, un message créé bloqué est envoyé au nœud contenant la fonction "+". Ceci montre l'asynchronisme qu'il y a entre les trois activités parallèles du modèle i.e l'exploration, la constitution des redex et la réduction.

A la fin de l'exploration de chaque paramètre, un message de fin d'exploration est envoyé à destination du message d'activation créé bloqué à destination du nœud "+". Une fois que les deux messages sont reçus, un message d'activation effectif est envoyé au nœud "+". La prise en compte de ce message implique l'envoi d'un message NENR à destination du nœud racine de l'argument. Le traitement de ce message est fait par échange de messages NRNR entre les trois nœuds de l'argument de la façon suivante comme le montre la figure 2.6.(d)). Le nœud racine envoie un message NRNR contenant la fonction "+" au premier nœud de la séquence. Ce dernier envoie un autre message NRNR à son successeur. Le message contient la fonction "+", le nom du nœud cible (racine) et la valeur contenue dans le nœud expéditeur. Le traitement du message consiste à additionner la valeur contenue dans le message i.e. "0" et celle contenue dans le nœud qui reçoit le message i.e. "1". Etant donné que le nœud qui contient la valeur "1" est le dernier de la séquence alors celui-ci envoie un message NRNR au nœud racine dans lequel il inclut le résultat de la fonction "+". L'arbre de réduction, argument du programme, est réduit à un seul nœud (figure 2.6.(e)) qui contient le résultat du programme.

#### 2.3.1.3 Mécanisme de copie

Certaines fonctions telle que la composition de fonctions "{}" (exemple précédent) nécessitent une copie d'arguments. Dans le simulateur, le mécanisme de copie est mis en œuvre en utilisant des messages de type "NRNR". La copie s'effectue en parallèle pour permettre de débloquer la réduction sur ces copies le plus vite possible. Une optimisation de cette méthode est possible en faisant un placement groupé des données i.e en ne faisant qu'une seule communication pour un groupe de nœuds de donnée adjacents placés sur un même processeur. Dans [BM95], la répartition dynamique des données est déclenchée et guidée par le mécanisme de copies. Ce qui permet une rectification du placement des données de façon à améliorer le degré de localité et celui du parallélisme.

#### 2.3.1.4 Topologie

La machine considérée dans la simulation est constituée d'un ensemble de sites (processeur+mémoire locale+couche de communication) reliés par un réseau d'interconnexion. Chaque site comprend une unité d'exploration et une unité de réduction pour le traitement respectivement des messages d'exploration et des messages de réduction. Ces deux unités partagent un espace mémoire local. Un site comprend également un ensemble de files d'attente de messages de différents types. Avant d'être orientés par un répartiteur local vers ces files d'attente, les messages provenant de l'extérieur sont déposés dans une file globale. Les messages émis à destination d'autres sites sont déposés dans une zone tampon gérée par un routeur dont le rôle est d'orienter les messages vers leurs destinataires respectifs. Les structures des différentes unités, des files d'attentes et leur gestion sont détaillées dans [Ben94].

#### 2.3.1.5 Parallélisme

Le parallélisme, dans le modèle simulé, est exploité en échangeant et en traitant les messages, tous types confondus, en parallèle. Les problèmes posés par les implémentations parallèles des langages fonctionnels cités dans le chapitre 1 apparaissent de la façon suivante :

- Le problème de synchronisation se pose principalement dans les situations suivantes :
  - L'exploration du successeur d'un nœud forme fonctionnelle ne peut se faire que si les sous arborescences fonctionnelles paramètres sont explorées.
  - L'exploration du successeur d'un nœud définition ne peut se faire que si l'arborescence fonctionnelle représentant la définition est entièrement explorée.

  Il existe d'autres situations posant le problème de synchronisation telle que la réalisation dynamique des copies.

- Le problème d'ordonnancement se pose lors du traitement des messages se trouvant dans les différentes files d'attente d'un même site. Il se pose particulièrement pour les messages de réduction ayant le même nœud cible. Un algorithme d'ordonnancement est proposé dans [Dev90] et dans [Ben94].

- La régulation de charge et la localité sont deux problèmes qui concernent essentiellement les données. Un algorithme de répartition à base d'un seuil de regroupement est décrit dans [Ben94, BM95].

- L'unité de traitement dans la simulation est le message. Par conséquent, la granularité de parallélisme est très fine puisque le traitement d'un message consiste en général à créer ou envoyer d'autres messages. Un regroupement des nœuds implique nécessairement le regroupement des messages qui leur sont destinés. L'algorithme de répartition est donc un moyen de grossissement de la granularité.

- La collecte de miettes (ici de nœuds) est faite en utilisant des messages de service dits "messages de destruction". La destruction des nœuds s'effectue en parallèle avec l'exécution du programme.

### 2.3.2 Résultats de simulation

Les tests de simulation du modèle ont été faits dans [Ben94] sur deux types de programmes : les programmes fortement récursifs (le quicksort sur 30 entiers et la suite de fibonacci 14) et les programmes non récursifs et manipulant des données de grande taille (la multiplication de 2 matrices carrées de dimension 10 et la multiplication de polynômes de taille 10). La table ci-dessous montre les résultats théoriques maximaux obtenus. Cette table est empruntée à la thèse [Ben94]. Le temps de simulation utilisé est un temps logique, il est mesuré en Ut (Unité de temps). Le rapport considéré entre le temps de communication (d'un message) et le temps de traitement (d'un message) est égal à 2. Le taux de parallélisme représente le nombre de processeurs actifs simultanément. Le nombre de processeurs de la machine étant considéré potentiellement infini.

| *programmes* / *résultats* | *Quicksort* | *Fibonacci* | *mul. matrices* | *mul. polynômes* |
|---|---|---|---|---|
| temps de simulation | 11862 Ut | 234 Ut | 272 Ut | 188 Ut |
| nb. total de messages | 61007 Mes. | 36127 Mes | 20026 Mes | 36135 Mes. |
| taux de parallélisme | 5.14 | 154.39 | 73.62 | 192.21 |
| messages traités / Ut | 5.14 Mes. | 154Mes. | 73 Mes. | 192 Mes. |
| nb. de noeuds créés | 12368 nds | 6398 nds | 3636 nds | 6859 nds |
| nb. de noeuds détruits | 12178 nds | 6331 nds | 386 nds | 110 nds |
| nb. de copies [a] | 1796 | 2635 | 100 | 199 |
| nb. max de noeuds copiés | 31 nds | 1 nds | 223 nds | 131 nds |
| nb. moyen de noeuds copies | 12 nds | 1 nds | 11 nds | 25 nds |
| Messages ACT traités | 5218 Mes. | 6022 Mes. | 340 Mes. | 1443 Mes. |
| Messages FE traités | 2088 Mes. | 3385 Mes. | 105 Mes. | 1100 Mes. |
| Messages ACK traités | 1352 Mes. | 2634 Mes. | 115 Mes. | 911 Mes. |
| Messages NENR traités | 5725 Mes. | 6398 Mes. | 1529 Mes. | 3112 Mes. |
| Messages NRNR traités | 12617 Mes. | 4891 Mes. | 9787 Mes. | 12504 Mes. |
| Messages de service traités | 34007 Mes. | 12797 Mes. | 8150 Mes. | 17065 Mes. |

[a] Le nombre maximum de noeuds copiés, donné dans ce tableau, est mesuré par copie

Les résultats de simulation illustrés par la table ci-dessus sont commentés dans le paragraphe suivant. Ils nous permettront de poser la problématique de la thèse et de fixer les objectifs de celle-ci.

## 2.4 Problématique et objectifs

Les résultats de simulation ci-dessus révèlent que le taux de parallélisme théorique[4] obtenu (nombre de messages traités par unité de temps) est important. Ceci traduit l'importance du parallélisme implicite possible dans le modèle $P^3$.

Les résultats mettent également en évidence un certain nombre de problèmes dont voici les principaux.

- Le nombre de nœuds détruits est important, particulièrement pour les programmes récursifs. Ceci montre la nécessité de disposer d'un outil performant de collecte de miettes (nœuds).
- Le nombre de copies effectuées est important pour les programmes récursifs et leur volume l'est pour les autres programmes. Par conséquent, une mauvaise gestion des copies induirait un surcoût de communication et de synchronisation. C'est pourquoi, le mécanisme de copie doit être optimisé.
- Le nombre de messages d'exploration et de réduction est considérable. Ceci soulève, entre autres, deux grands problèmes :
  - Le traitement de chaque message d'exploration, plus exactement d'activation, nécessite d'effectuer un test pour prendre connaissance de la fonction contenue dans le nœud fonctionnel activé. Ainsi, une approche d'implantation interprétée induirait un surcoût de calcul énorme. Une approche compilée est donc souhaitable.
  - Le message étant l'unité de charge, la granularité est donc très fine. Ceci a pour avantage de pouvoir équilibrer la charge mais présente un grand inconvénient, celui d'induire un surcoût de communication exhorbitant. Aussi, un effort conséquent est indispensable pour regrouper les nœuds afin de limiter les communications. Cet effort traduit la prise en charge du problème de granularité.

    ~~Le regroupement des nœuds (ou grossissement de la granularité) apparaît à trois~~ niveaux :
    - Dans l'espace d'exploration : regroupement des nœuds fonctionnels pour minimiser le surcoût occasionné par les messages d'exploration ;
    - Dans l'espace de réduction : regroupement des nœuds de données pour limiter le surcoût entraîné par les messages de type NRNR ;
    - Entre les deux espaces i.e. regrouper les nœuds fonctionnels agglomérés avec leurs nœuds de données cibles respectifs. Ceci permettra de réduire le surcoût induit par les messages de type NENR.

[4] nombre de processeurs potentiellement infini

Les résultats de simulation rapportés au paragraphe précédent n'illustrent pas directement le problème de régulation de charge. D'autres résultats issus de l'étude de ce problème sont présentés dans [Ben94, BM95]. Cette étude est centrée sur les activités de copie d'arguments. Un mécanisme basé sur un seuil de regroupement est utilisé pour corriger dynamiquement la répartition des données. Il constitue également un outil de grossissement de la granularité.

Les problèmes abordés dans la thèse ainsi que les objectifs de celle-ci sont présentés ci-dessous.

### 2.4.1 Granularité de parallélisme

Le premier objectif de la thèse est de proposer une approche de la gestion de la granularité de parallélisme. Une telle approche doit répondre aux trois points suivants :

- Partitionnement du programme en paquets de fonctions selon des critères à définir ;
- Définition, de façon abstraite, pour chacun des paquets de fonctions, des données dont il a besoin. On appellera ces données agglomérées *une fenêtre*. Ce point exprime un regroupement "abstrait" des données, qui se concrétisera à l'exécution ;
- Regroupement des fonctions et des données par association des paquets de fonctions et de leurs fenêtres.

Cette approche permettra de répondre également au problème d'optimisation des copies. En effet, un mécanisme de copie paresseuse est (indirectement) visé.

### 2.4.2 Régulation dynamique de charge

Le problème de régulation de charge concerne deux types d'entités : les associations paquet-fenêtre ainsi que les données ne pouvant pas tenir sur le même site.

Le deuxième objectif de la thèse est de proposer une politique de répartition de ces entités qui :

- soit générale (réutilisable) i.e. qui ne soit pas liée au modèle $P^3$ ;
- induise un surcoût de calcul et de communication le moindre possible ;
- améliore le temps global d'exécution des programmes.

### 2.4.3 Approche d'implantation compilée et multithreadée

Le troisième objectif de la thèse est de proposer une approche d'implantation du modèle $P^3$ en utilisant une approche compilée et à base de threads. L'utilisation du multithreading a pour but de réduire davantage le surcoût occasionné par les communications. Après transformation d'un programme, celui-ci doit être traduit dans un langage compilable et intégrant une bibliothèque de primitives de gestion de threads dans un contexte distribué. L'environnement C+$PM^2$ est utilisé à cet effet.

La définition d'un ramasse-miettes est un objectif à long terme de cette thèse.

## 2.5 Conclusion

Dans ce chapitre, nous avons décrit brièvement le modèle $P^3$. Celui-ci est basé sur une représentation séparée du programme et de sa donnée. Le programme est représenté par une forêt d'arborescences fonctionnelles. Les données sont contenues dans des arbres binaires. L'évaluation d'un programme fonctionnel selon $P^3$ utilise deux mécanismes : le mécanisme d'exploration et le mécanisme de réduction. Ces deux activités sont exécutées en parallèle. Une simulation à base de messages du modèle a révélé ou confirmé un certain nombre de problèmes. Parmi ces derniers, on distingue les problèmes de granularité, de régulation dynamique de charge et d'utilisation d'une approche d'implantation interprétée. Ces trois principaux problèmes constituent la problématique de cette thèse.

# Partie II

# Granularité de parallélisme

# Chapitre 3

# Granularité de parallélisme

## 3.1 Introduction

Les processus créés lors de l'exécution parallèle d'un programme occasionnent un surcoût par leur création, leur exécution (changements de contexte) et leur terminaison. Par souci d'efficacité, ce surcoût doit être très faible en comparaison de la quantité de travail que doit effectuer chaque processus. Par conséquent, le programme doit éviter de générer des processus dont le coût de gestion est, a priori, supérieur au travail effectif qu'il est censé effectuer. En d'autres termes, il est nécessaire de grossir la taille des processus du programme et ce, au détriment de la concurrence. Le choix de la bonne taille des processus est crucial. En effet, si celle-ci est petite alors le surcoût associé à l'administration des processus est exhorbitant. A l'inverse, si celle-ci est grande alors il y aura une perte de parallélisme. Ce problème est dit *problème de granularité.*

La notion de granularité est intuitivement claire mais jusqu'à cette date il n'existe pas de définition rigoureuse de celle-ci. La définition dépend de la nature et de la taille du programme ainsi que de l'architecture supportant son exécution. Deux définitions sont généralement utilisées : la première définit la granularité de parallélisme comme étant le ratio $e/c$, où $e$ désigne le temps passé par un processus dans le calcul et $c$ son temps de communication. La deuxième définition est celle qui considère que la granularité est proportionnelle au nombre de fois qu'un processeur passe du calcul à la communication.

Très peu de techniques de gestion du problème de granularité ont été proposées dans la littérature. Parmi celles qui existent, on distingue, selon l'implication du programmeur dans la gestion de la granularité, les *méthodes explicites* et les *méthodes implicites.* Alors que les méthodes explicites laissent le fardeau de fixer la granularité au programmeur, les méthodes implicites gèrent la granularité automatiquement. Les méthodes explicites utilisent généralement des *annotations* [BHLH+93, Bur84]. Les méthodes implicites sont de deux types : celles qui utilisent une *analyse statique* [HG85] du programme par le compilateur et celles qui reposent sur une *gestion dynamique* [ABF93, Jon89, RS87, Rab91b] de la granularité.

La suite de ce chapitre est organisée de la façon suivante. Avant tout, nous faisons un bref tour d'horizon des méthodes explicites et implicites qui existent. Ensuite, nous indiquons comment le problème est traité dans les implémentations parallèles des langages fonctionnels présentées dans le premier chapitre.

## 3.2 Approche explicite : les annotations

Dans l'approche explicite, le contrôle de la granularité est laissé entièrement à la charge du programmeur. Il est fait explicitement à l'écriture du programme en ajoutant des annotations. Ces dernières sont des directives permettant de spécifier les points de génération de parallélisme. Les indications "Future"[MKH91] et "par" [THM+96] sont deux exemples d'annotations.

Certaines méthodes utilisant l'approche explicite attachent aux expressions fonctionnelles du programme des conditions de génération de tâches pour les réduire. Ces conditions utilisent généralement un seuil. Ces méthodes sont très adaptées aux problèmes de type "diviser pour régner" [RM91, THM+96, BHLH+93]. Cependant, celles-ci présentent deux grands problèmes. Premièrement, le seuil est dépendant de l'architecture d'exécution utilisée. Deuxièmement, il n'est pas toujours facile d'exprimer la condition associée à une expression. En effet, il suffit, comme on peut le voir à travers un exemple donné dans [RM91], de considérer une condition (ou expression) comportant plusieurs variables.

## 3.3 Approche implicite

L'approche implicite est une approche automatique. Elle utilise soit une analyse statique du programme à la compilation ou une gestion dynamique à l'exécution du programme.

### 3.3.1 Analyse statique

Les limitations de l'approche explicite font qu'il est préférable que les annotations soient insérées dans le programme par le compilateur plutôt que par le programmeur. Cette idée a été utilisée par Goldberg et Hudak dans [HG85] pour définir la notion de *combinateurs sériels* (pseudo-fonctions). Le principe de ces combinateurs repose sur l'analyse statique de la complexité des expressions fonctionnelles (pièces de code). Celle-ci permet d'identifier les expressions pour lesquelles des processus doivent être créés.

L'idée des combinateurs sériels est la plus largement référencée. Cependant, elle présente des problèmes. En effet, en présence de variables libres et de fonctions récursives, une complexité infinie est associée à une expression alors que le coût de l'expression peut s'avérer négligeable à l'exécution. L'idée a été expérimentée dans [GH86]. Elle s'est révélée peu efficace.

### 3.3.2 Gestion dynamique

L'approche dynamique est très utilisée dans les implémentations des langages fonctionnels car elle tient plus compte du caractère dynamique et irrégulier de ces langages au prix d'un surcoût de calcul et de communication. Par opposition aux approches précédentes, l'approche dynamique réalise une gestion de la granularité en tenant compte d'un certain nombre d'informations déterminées à l'exécution du programme. Ces informations peuvent être liées au programme comme elles peuvent être liées à la machine.

Dans [RS87], le mécanisme d'étranglement contrôle la granularité en tenant compte de l'état de charge de la machine. Si la machine est en surcharge alors on arrête de générer des tâches. Chaque tâche continue son exécution en séquentiel. Dans le modèle "évaluer et mourir" [Jon89], un processus père effectue le travail qu'il a délégué à son processus fils si ce dernier ne l'a pas encore commencé. Ce qui permet de grossir la granularité du processus père.

Une méthode utilisant les caractéristiques du programme est celle définie dans [ABF93]. Cette méthode est applicable aux programmes représentés par des arbres tels que les programmes de type "diviser pour régner". Elle est itérative et à chaque itération, le critère de gestion de la granularité utilisé est la vitesse à laquelle sont générées les tâches à l'itération précédente.

Dans [Rab91b], une généralisation du concept des combinateurs sériels est proposée. La méthode utilise deux types d'analyse. La première est une analyse statique de la *complexité séquentielle*. La complexité séquentielle d'une expression fonctionnelle est proportionnelle au nombre d'appels de fonctions utilisées dans cette expression. Le deuxième type d'analyse, qui permet un contrôle dynamique de la granularité, est la *complexité de tâche*. La complexité de tâche se détermine par la structure générale des tâches, leur nombre et leurs tailles. D'autres types d'informations peuvent être inclues pour assurer un contrôle dynamique de la granularité comme par exemple la charge des machines. Les résultats expérimentaux présentés dans [Rab91b] montrent que l'utilisation de la complexité séquentielle est peu pratique. La complexité de tâche est plus adaptée aux problèmes de type "diviser pour régner". De plus, elle dépend de ce que le problème à traiter est régulier ou non et équilibré ou non.

Dans le paragraphe suivant, nous allons montrer comment le problème de granularité est géré dans les exemples d'implémentations parallèles des langages fonctionnels présentées dans le chapitre 1.

## 3.4 Exemples

La machine MaRS est normalement conçue pour des applications à granularité fine. Cependant, les expressions dont la parallélisation est jugée inefficace à la compilation sont séquentialisées. Le mécanisme de *continuation* [CDG+86] est utilisé à cet effet. Dans ce mécanisme, une sous-expression dont le coût de réduction est faible est exécutée par le même processus qui exécute l'expression qui la contient. La granularité est dans ce cas gérée implicitement.

Une gestion explicite de la granularité est également possible dans MaRS. Deux annotations [Cou91] sont utilisées : "processus" pour générer un processus de réduction et "Sequent" pour séquentialiser un traitement.

Dans Flagship, comme cela a été dit dans le chapitre 1, l'unité d'exécution et d'allocation est un paquet exécutable. Flagship utilise le mécanisme d'étranglement [Sar87] pour gérer la granularité.

Dans Rediflow, les nœuds du graphe représentant un programme selon le modèle Rediflow encapsulent chacun plusieurs instructions de réduction. Ainsi, chaque nœud peut consommer plusieurs données. Ce qui permet de grossir la granularité. Par ailleurs, comme dans Flagship le mécanisme d'étranglement est également utilisé. En mode séquentiel, plusieurs nœuds du graphe peuvent être traités par le même processus.

Dans GRIP, les unités d'exécution et d'allocation sont des tâches. Chaque tâche est créée pour réduire un sous-graphe. D'une part, l'annotation "par" est utilisée pour indiquer les expressions pour lesquelles il faudra créer des tâches. De ce fait, la gestion de la granularité est explicite. D'autre part, le modèle "évaluer et mourir" de synchronisation de tâches est utilisé comme moyen pour grossir la granularité de parallélisme. Dans ce cas, il s'agit d'une gestion implicite de la granularité.

Dans la machine GUM, la granularité est gérée de la même façon que dans GRIP sauf que l'unité d'exécution et d'allocation considérée est un thread au lieu d'une tâche.

## 3.5 Conclusion

Dans ce chapitre, nous avons identifié les différentes approches de gestion de la granularité. On distingue principalement les méthodes explicites et les méthodes implicites. Les approches explicites reposent sur l'utilisation d'annotations au niveau du programme. Par contre, les approches implicites laissent le fardeau de la gestion du problème de granularité au compilateur. Deux types de méthodes implicites ont émergé : celles qui effectuent un partitionnement statique à la compilation et celles qui assurent un contrôle dynamique de la granularité à l'exécution. L'inconvénient des méthodes statiques est qu'elles ne tiennent pas compte du caractère dynamique des langages fonctionnels. A l'inverse, les approches dynamiques ont le défaut d'induire un surcoût de gestion supplémentaire. Le chapitre montre également comment le problème de granularité est traité dans les machines présentées dans le chapitre 1.

Le chapitre suivant décrit l'approche de gestion de la granularité que nous avons proposée dans le cadre du modèle $P^3$.

# Chapitre 4

# Une approche de la gestion de la granularité

## 4.1 Introduction

Les langages fonctionnels se caractérisent par une granularité fine. Cela constitue un "handicap" pour l'exploitation de tout le parallélisme potentiel qui leur est inhérent. Ce caractère de la granularité a été confirmé dans le modèle $P^3$ à travers une simulation à base de messages [Ben94].

Comme il a été noté dans le chapitre 2, le grossissement de la granularité dans le modèle $P^3$ nécessite un regroupement à trois niveaux. Le premier regroupement concerne les fonctions du programme tandis que le deuxième se rapporte aux nœuds de données. Le troisième niveau de regroupement est une association des différentes partitions du programme avec leurs données regroupées correspondantes.

Dans ce chapitre, nous proposons une nouvelle approche implicite de gestion de la granularité. Le but principal de cette approche est de réaliser les trois types de regroupement cités ci-dessus en minimisant la perte de parallélisme.

Afin d'effectuer le premier type de regroupement (de fonctions), le programme subit une transformation. La première étape de cette transformation consiste à identifier tout le parallélisme horizontal et vertical possible. Les critères de cette identification doivent être définis. Les éléments de programme obtenus par ce partitionnement sont appelées *tronçons.*

Pour chaque tronçon, on détermine de façon "abstraite" la donnée minimale recouvrant la donnée dont il a besoin pour s'exécuter. Cette donnée est appelée *fenêtre cible.* Ceci constitue un regroupement implicite des données. Les tronçons et leurs fenêtres cibles constituent des *segments.*

Les segments sont ensuite regroupés afin d'éliminer le parallélisme jugé, a priori, inefficace. A cette étape, des critères de limitation de parallélisme doivent être également définis. Les nouvelles partitions obtenues sont appelées des *paquets.*

La partie code de chaque paquet est traduite en langage C couplé avec une bibliothèque de primitives de gestion de threads dans un contexte distribué. Chaque paquet devient ainsi un

*module*. Chaque module devient un thread à l'exécution. L'association du code d'un module et de sa fenêtre cible constitue le troisième niveau de regroupement.

La suite du chapitre est organisée de la manière suivante. Avant tout, nous définissons dans le paragraphe 2 les différents concepts utilisés dans notre approche. Ensuite, nous décrivons brièvement dans le paragraphe 3 les différentes étapes de transformation subies par un programme Graal en vue de grossir sa granularité. Dans les trois paragraphes suivants, nous décrivons de façon détaillée chacune des étapes de la transformation. Ensuite, nous illustrons dans le paragraphe 6 les différentes étapes de la transformation sur un exemple de programme. Enfin, nous concluons le chapitre dans le paragraphe 7.

## 4.2 Concepts utilisés

Les concepts utilisés dans notre approche sont la *fenêtre*, le *tronçon*, le *segment*, le *paquet* et le *module*. La suite de ce paragraphe donne une définition de chacun de ces concepts.

### 4.2.1 Notion de fenêtre d'une fonction

**Définition 4.1 : La fenêtre cible d'une fonction (ou *FCF*)** est le sous-arbre de données, partie de son argument, susceptible d'être **effectivement utilisé** lors de l'exécution du code de cette fonction. La racine de la fenêtre est la racine de l'argument.

**Exemple** : La figure 4.1 montre la fenêtre cible associée à la fonction *cons* du langage Graal. On rappelle que la fonction *cons* ajoute un élément (x sur la figure) à une liste d'objets ($< y_1, y_2, ..., y_m >$ sur la figure). Les nœuds du sous-arbre représentant la fenêtre cible sont ceux encerclés dans la figure 4.1. Les autres nœuds demeurent inchangés.

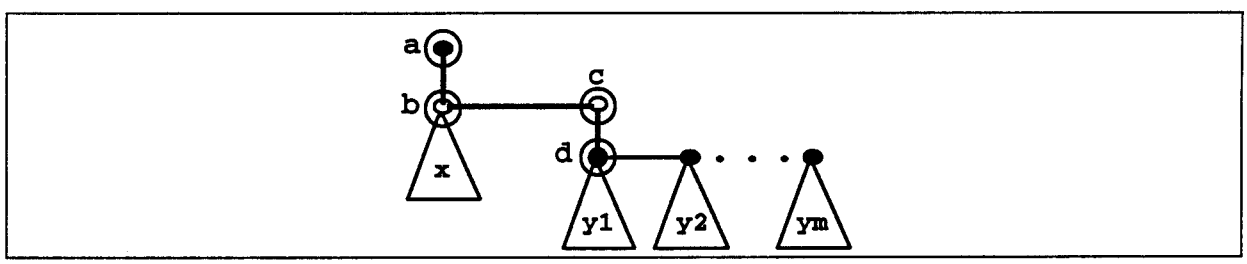



Figure 4.1 : Fenêtre cible de la fonction Cons

**Définition 4.2 : La fenêtre résultat d'une fonction (ou *FRF*)** est le sous-arbre de données résultant de l'exécution de la fonction sur sa fenêtre cible. La racine de la fenêtre résultat est la même que celle de la fenêtre cible.

**Exemple** : La figure 4.2 montre la fenêtre résultat associée à la fonction *cons* du langage Graal. Les nœuds du sous-arbre représentant cette fenêtre sont ceux encerclés dans la figure.

**Définition 4.3 : Le descripteur de la fenêtre cible ou DFCF (resp. résultat ou DFRF)**

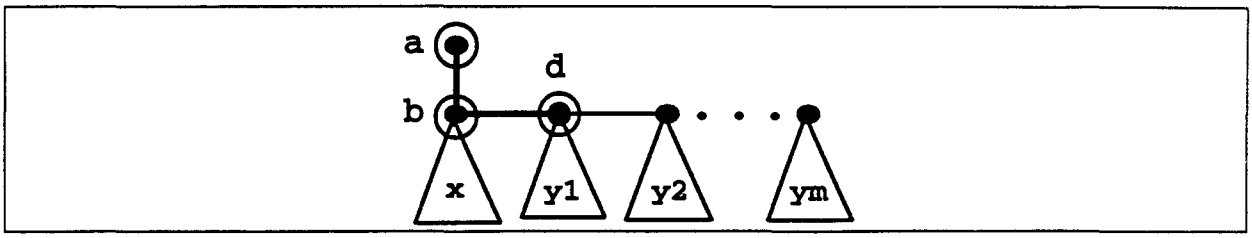



Figure 4.2 : Fenêtre résultat de la fonction Cons

**d'une fonction** décrit l'organisation de l'ensemble des nœuds constituant la fenêtre cible (resp. résultat) de cette fonction. C'est une représentation de la fenêtre (sous-arbre) dans un monoïde libre.
L'alphabet de ce monoïde est défini au paragraphe 4.5.2.

### 4.2.2 Notion de tronçon

**Définition 4.4** : Un **tronçon** est une séquence (composition) de fonctions (son exécution est de **nature séquentielle**). Il constitue une partie d'un chemin séquentiel. Formellement, un tronçon $t$, constitué de $k$ fonctions, peut s'écrire sous la forme suivante :

$$t = f_1 \, o \, f_2 \, o \, ... \, o \, f_k$$

**Définition 4.5** : La **fenêtre cible d'un tronçon (ou** $FCT$**)** est le sous-arbre de données susceptible d'être **effectivement utilisé** lors de l'application du code associé au tronçon.

**Définition 4.6** : La **fenêtre résultat d'un tronçon (ou** $FRT$**)** est le sous-arbre de données résultant de l'exécution du tronçon sur sa fenêtre cible.

**Définition 4.7** : Le **descripteur de la fenêtre cible ou DFCT (resp. résultat ou DFRT) d'un tronçon** est le codage de la fenêtre cible (resp. résultat) de ce tronçon dans un monoïde libre.

### 4.2.3 Notion de segment

**Définition 4.8** : Un **segment** est un couple $(t, d)$ constitué d'un tronçon $t$ et du descripteur $d$ de la fenêtre cible associée à ce tronçon.

### 4.2.4 Notion de paquet

**Définition 4.9** : Un **paquet** est une sous-arborescence de segments dont l'exécution est **décidée** séquentielle à la compilation.
Si un *paquet* est constitué de $k$ segments, il peut être écrit sous la forme :

$$\{s_1, s_2, ..., s_k\} = \{(t_1, d_1), (t_2, d_2), ..., (t_k, d_k)\}$$

où $s_i = (t_i, d_i)$ désigne le $i^{eme}$ segment du paquet.

***Remarque :*** L'ordre d'apparition des segments dans le paquet ne définit pas l'ordre d'exécution de ces derniers.

**Définition 4.10** : La **fenêtre cible d'un paquet (ou** $FCP$**)** est une forêt de sous-arbres de données susceptible d'être **effectivement utilisée** lors de l'exécution du code associé à l'ensemble des tronçons du paquet. En d'autres termes, la fenêtre cible d'un paquet donné est l'union des fenêtres cibles associées à ses différents tronçons. L'écriture formelle d'une $FCP$ est la suivante :
Soient $w_{t1}, w_{t2}, ..., w_{tk}$ les fenêtres cibles associées respectivement aux tronçons $t_1, t_2, ..., t_k$ d'un paquet donné. La fenêtre $w_p$ associée à ce paquet est la suivante :

$$w_p = \bigcup_{i=1}^{k} w_{ti}$$

**Définition 4.11** : La **fenêtre résultat d'un paquet (ou** $FRP$**)** est le sous-arbre de donnée résultant de l'exécution de l'ensemble des tronçons du paquet.

**Définition 4.12** : Le **descripteur d'une fenêtre cible d'un paquet ou DFCP** décrit l'organisation de l'ensemble des nœuds constituant la fenêtre cible de ce paquet. Autrement dit, c'est la collection des $DFCT$ des différents tronçons du paquet. Si un paquet est constitué de $k$ segments $\{(t_1, d_1), (t_2, d_2), ..., (t_k, d_k)\}$ et si le descripteur de sa fenêtre cible est appelé $D$ alors ce dernier peut s'écrire :

$$D = \{d_1, d_2, ..., d_k\}$$

***Remarque :*** Pour séparer sa partie fonctionnelle de sa partie donnée, un paquet peut s'écrire également sous la forme $(T, D)$ où T et D sont donnés ci-dessous :

$$T = \{t_1, t_2, ..., t_k\} \; et \; D = \{d_1, d_2, ..., d_k\}$$

### 4.2.5 Notion de module

**Définition 4.13** : Un **module** est un couple $(C, D)$ où $C$ est un code séquentiel et $D$ est un descripteur d'une fenêtre de données.

**Remarque** : Un paquet $(T, D)$ sous sa forme exécutable est un module. En d'autres termes, le couple constitué du code associé au paquet et de son $DFCP$ représente un module. Si le paquet est constitué de $k$ segments alors le module, noté $||T, D||$, associé à ce paquet est le couple $(C, D)$ tel que $C$ est le code obtenu par traduction des tronçons du paquet dans un environnement de programmation distribuée et par ajout de primitives de synchronisation de ces tronçons.

### 4.2.6 Illustration des relations entre les différents concepts

Le graphe de la figure 4.3 montre les relations entre les différents concepts du modèle. Les couples $\{(t_1, d_1), (t_2, d_2), ..., (t_k, d_k)\}$ constitués de *tronçons* et leurs $DFCT$ associés forment respectivement les *segments* $s_1, s_2, ..., s_k$. Ces segments sont regroupés pour former un *paquet*. La

partie fonctionnelle ($\{t_1, t_2, ..., t_k\}$) du paquet est appelée $T$ et son $DFCP$ est appelé $D$. Par codage de $T$ dans un environnement de programmation distribuée et par ajout de primitives de synchronisation, on obtient un code appelé $C$. Le couple $(C, D)$ est appelé *module.*

**Remarque** : Il existe un autre type de module, issu de l'exploitation du parallélisme de données. Un exemple d'un tel module est l'application d'une fonction polyadique à une partition d'une longue séquence de données. Dans ce cas, la partie code du module est le code de la fonction polyadique et la partie donnée est le descripteur associée à une séquence partielle de données. Les constituants de ce type de module sont représentés par des traits en pointillé sur la figure 4.3.

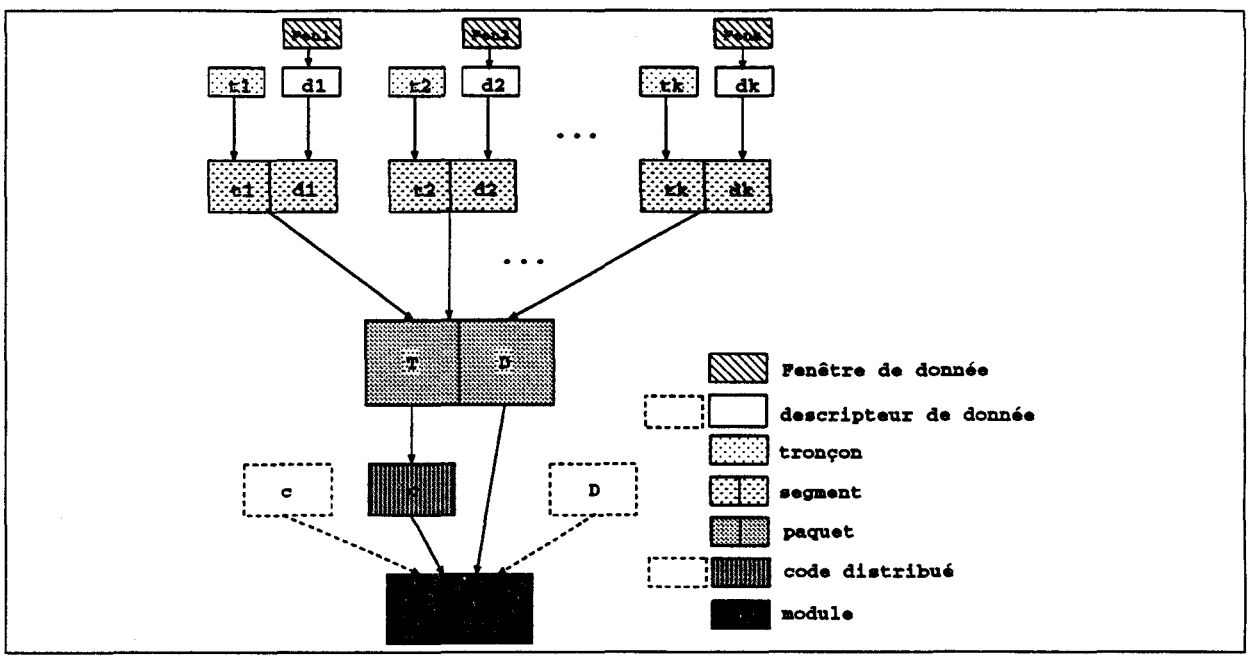



Figure 4.3 : Les relations entre les différents concepts

## 4.3 Transformation de programme

### 4.3.1 Etapes de transformation

Avant d'être exécuté, un programme subit une transformation. Le but de cette transformation est de grossir la granularité de parallélisme de l'exécution du programme au prix d'une perte de concurrence (celle dont l'exploitation n'est, a priori, pas efficace). Les étapes de l'approche proposée pour le grossissement de la granularité sont les suivantes :

- **Découpage du programme en tronçons** : A cette étape, le programme est reçu par un algorithme dit *de découpage* sous forme d'une forêt d'arborescences de fonctions. Il

est ensuite découpé en tronçons. Les objectifs du découpage et les critères utilisés sont définis dans la section 4.4. A l'issue de cette étape, on obtient une forêt d'arborescences de tronçons.

- **Calcul du descripteur de fenêtre *DFCT* associé à chaque tronçon du programme** : Pour chacun des tronçons issus du découpage effectué à l'étape précédente, on calcule le *DFCT* associé. Les tronçons couplés avec leurs *DFCT* associés forment des segments. La section 4.5 présente l'alphabet de description utilisé ainsi que l'outil de calcul des *DFCT*. A l'issue de cette étape, on obtient une forêt d'arborescences de segments.

- **Regroupement des segments du programme en paquets** : Les segments constitués aux deux étapes précédentes sont agglomérés pour former des paquets. Les motivations et les critères du regroupement sont présentés dans la section 4.6. Cette étape fournit en sortie une forêt d'arborescences de paquets.

- **Traduction des arborescences de paquets dans un langage distribué** : Les arborescences de paquets obtenues à l'étape précédente sont traduites dans un code distribué. A l'issue de cette étape, une bibliothèque de primitives est générée. Chaque primitive correspond au code associé à un paquet du programme. Bien que cette phase fasse partie du pré-traitement du code, nous préférons la développer dans le chapitre suivant car elle repose sur le mécanisme d'exécution (synchronisation, ordonnancement, ...) du programme.

La figure 4.4 résume les différentes étapes de transformation (grossissement de la granularité) d'un programme ainsi que l'outil qu'il est nécessaire de mettre en place à chacune de ces étapes.

**Remarque importante** : En présence de fonctions d'ordre supérieur dans le programme, les transformations ci-dessus doivent se faire également sur les arborescences fonctionnelles apparaîssant dans la partie argument du programme.

### 4.3.2 Notion de programme transformé

**Définition 4.14** : On appelle **programme transformé** tout programme Graal ayant passé les quatre étapes de transformation décrites ci-dessus. Un *programme transformé* peut être vu comme une collection de modules. Si, par exemple, il contient $m$ modules $M_1, M_2, ..., M_m$ alors on peut l'écrire sous la forme suivante :

$$\{M_1, M_2, ..., M_m\} = \{||T_1, D_1||, ||T_2, D_2||, ..., ||T_m, D_m||\}$$

## 4.4 Découpage d'un programme en tronçons

Nous rappelons qu'il existe deux formes de parallélisme : le parallélisme horizontal et le parallélisme vertical (cf. chapitre1). Ces deux formes sont illustrées par l'exemple de la figure 4.5. Cette figure montre l'application du programme fonctionnel ($g$ *o* (*dista list* $f$)) (fig. 4.5 a) à la donnée $< a_1, a_2, ..., a_n >$ (fig. 4.5 b).

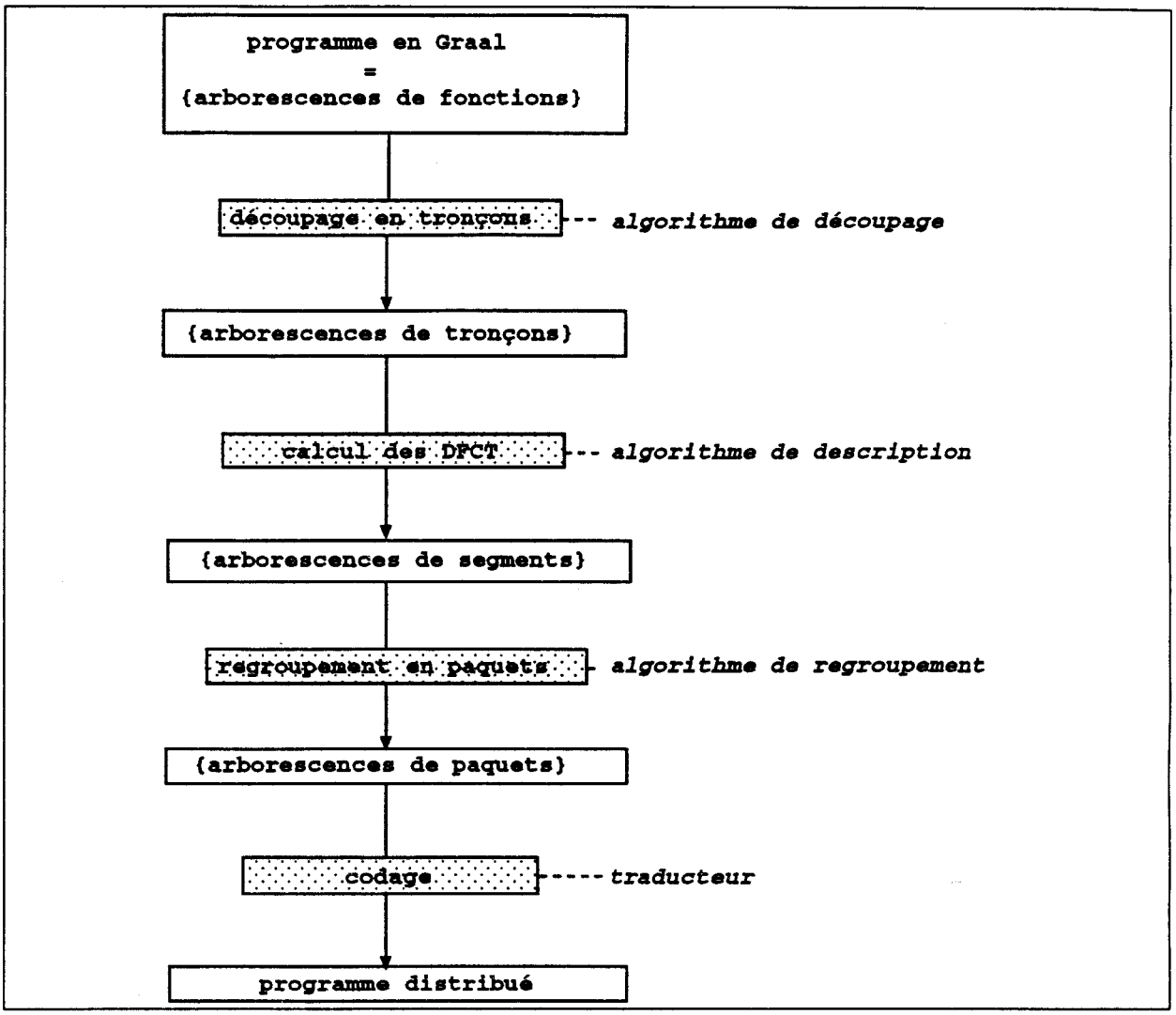



Figure 4.4 : Etapes de grossissement de la granularité d'un programme

Dans cet exemple, le parallélisme horizontal est exprimé par la forme fonctionnelle *dista*. En effet, les applications de la fonction $f$, deuxième argument de la forme fonctionnelle, sur les données $a_1, a_2, ..., a_n$ peuvent être exécutées en parallèle.

Le parallélisme vertical est produit par la présence de la fonction non stricte *list* dans le chemin séquentiel du programme. En effet, cette fonction peut s'exécuter avant la fin des applications de la fonction $f$ car son application utilise uniquement la racine de la séquence de données. De ce fait, la fonction $g$ peut s'exécuter en parallèle avec (*dista list* $f$). Ces deux parties du programme sont séparées par un trait en pointillé sur la figure.

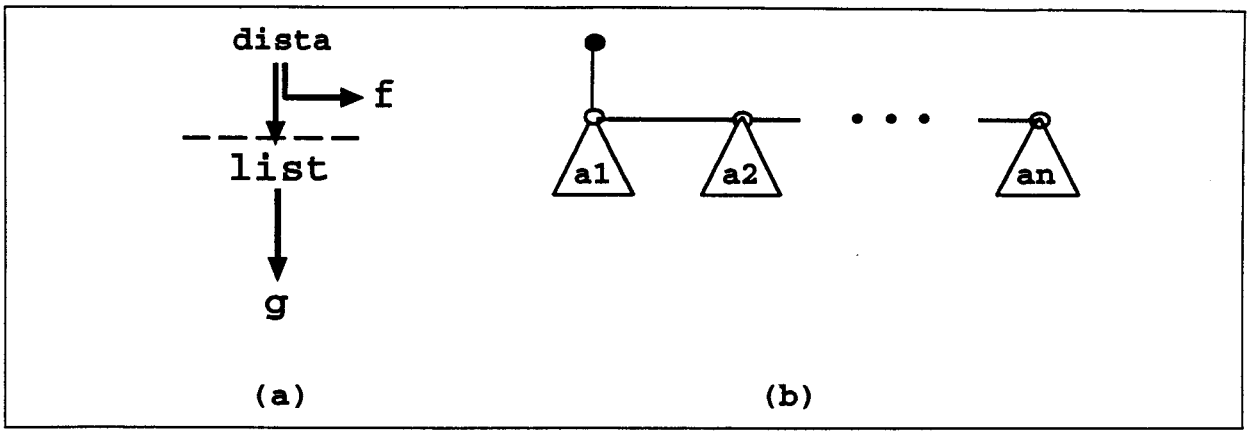



Figure 4.5 : Exemple illustratif des deux formes de parallélisme

### 4.4.1 Objectifs et critères

Le but du découpage d'un programme en tronçons est d'identifier tout le parallélisme horizontal et vertical **exploitable** intrinsèque à ce programme. Ce parallélisme identifié représente la limite théorique du parallélisme de traitement exploitable.

Avant de définir les critères de découpage d'un programme en tronçons, définissons d'abord la notion de *fonction frontière.*

**Définition 4.16** : une **fonction frontière** est une fonction qui définit le début d'un tronçon. Elle peut être :

- une fonction non stricte ;
- la racine d'un paramètre d'une forme fonctionnelle ;
- une fonction successeur d'une forme fonctionnelle génératrice de parallélisme (ex. $\alpha$, [ ], ...) ;
- une fonction successeur d'une fonction d'ordre supérieur.

Le découpage d'un programme en tronçons est une simple identification de toutes les fonctions frontières présentes dans les arborescences fonctionnelles du programme. Ces fonctions frontières représentent les critères de découpage. Elles permettent de délimiter tous les tronçons du programme. Ci-dessous, nous examinons chacun des différents types de fonctions frontières et nous donnons dans chacun des cas de figure un exemple illustratif du découpage résultant de la présence dans le programme du type de fonction frontière en question.

#### 4.4.1.1 Cas d'une fonction non stricte

Une fonction non stricte permet d'identifier le parallélisme vertical dans un programme. En effet, il est possible d'anticiper sur l'exécution d'un tronçon commençant par une fonction non stricte, étant donné que celle-ci n'a pas besoin d'attendre le résultat de l'évaluation de son (ou ses) argument(s) pour s'exécuter.
L'exemple de la figure 4.5 illustre le cas où le critère de découpage (fonction frontière) est une fonction non stricte. En effet, la fonction non stricte *list* est une fonction frontière qui permet d'identifier le tronçon (*g o list*). L'anticipation sur l'exécution de la fonction *list* peut améliorer considérablement l'efficacité de l'exécution. Effectivement, si la fonction *g* est par exemple une fonction d'ordre supérieur, l'arborescence fonctionnelle que celle-ci doit produire sera construite en parallèle avec l'exécution de la forme fonctionnelle *dista f*. Si la fonction *g* se trouve à la fin d'un chemin séquentiel principal d'une définition alors le programme appelant sera débloqué.

#### 4.4.1.2 Cas de la racine d'un paramètre d'une forme fonctionnelle

Une fonction racine d'un paramètre d'une forme fonctionnelle est un outil d'identification du parallélisme horizontal. Les fonctions *cdr* et *car*, racines respectivement du premier et deuxième paramètres de la forme fonctionnelle construction, de la figure 4.6 sont des fonctions frontières. Elles délimitent les tronçons racines de ces paramètres.

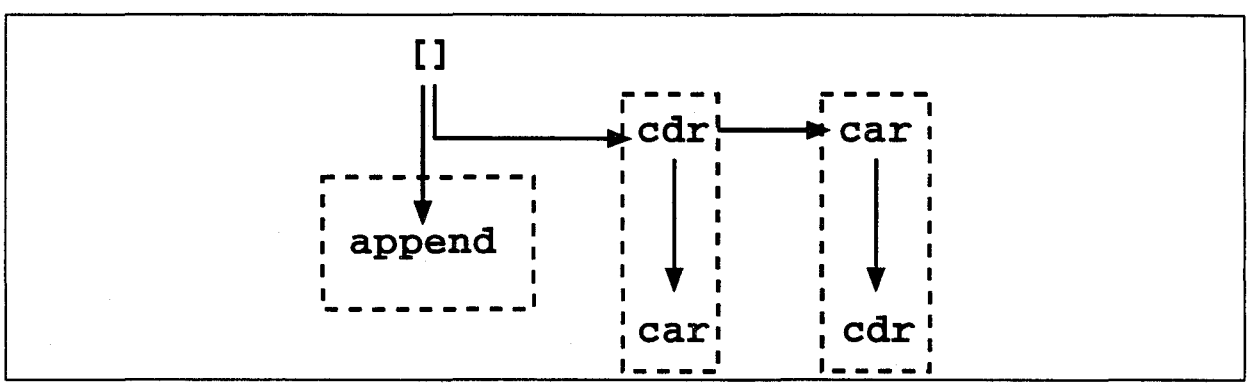



Figure 4.6 : Cas où la fonction frontière est le début d'un paramètre ou un successeur d'une forme fonctionnelle

#### 4.4.1.3 Cas d'une fonction successeur d'une forme fonctionnelle

Une fonction successeur d'une forme fonctionnelle doit être une fonction frontière pour deux raisons. Premièrement, la forme fonctionnelle peut produire du parallélisme (ex. $\alpha$, [ ], ...). Deuxièmement, on ne sait pas calculer, dans le cas général, le descripteur de la fenêtre cible associée à une forme fonctionnelle regroupée avec ses paramètres.
Un exemple illustrant ce cas de figure est donné dans la figure 4.6. Dans cette figure, la fonction *append* commence un nouveau tronçon car elle est successeur de la forme fonctionnelle

*construction.*

#### 4.4.1.4 Cas d'une fonction successeur d'une fonction d'ordre supérieur

Une fonction successeur d'une fonction d'ordre supérieur est une fonction frontière. La raison est que l'arborescence fonctionnelle constituée par une fonction d'ordre supérieur peut être elle-même décomposable et ceci ne peut être su qu'à l'exécution du programme ; or, on sait qu'un tronçon n'est pas décomposable. Par exemple, dans le programme illustré par la figure 4.7, le combinateur *comp* produit la forme fonctionnelle *composition* qui est une fonction génératrice de parallélisme. Par conséquent, la fonction $f_2$ ne doit par être dans le même tronçon que la fonction *comp*. Elle doit donc commencer un nouveau tronçon.

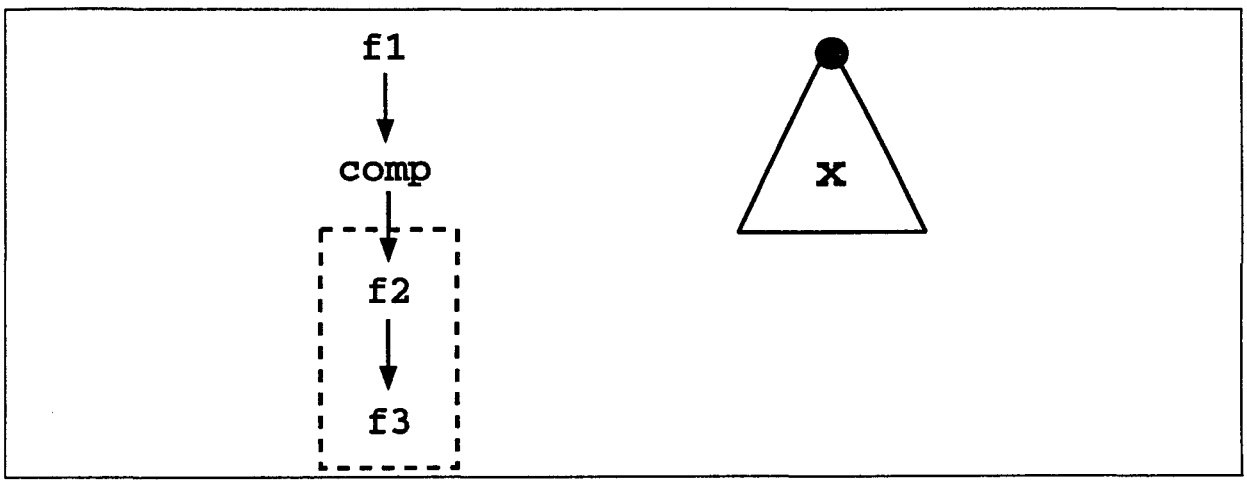



Figure 4.7 : Cas où la fonction frontière est un successeur d'une fonction d'ordre supérieur

## 4.5 Description de la fenêtre cible d'un tronçon

Nous nous proposons, dans cette section, de déterminer le descripteur associé à la fenêtre cible d'un tronçon donné. Avant de présenter l'outil de description nous poserons dans le paragraphe suivant la problématique et nous donnerons également le principe de description.

### 4.5.1 Problématique et principe de description

Le problème du calcul de la fenêtre cible d'un tronçon peut être exprimé de la façon suivante : Soient $t$ un tronçon de fonctions et $w_t$ et $r_t$ respectivement sa fenêtre cible et sa fenêtre résultat. On cherche à déterminer le descripteur $Dw_t$ associé à la fenêtre $w_t$.

Avant d'associer un descripteur à la fenêtre cible d'un tronçon, examinons d'abord le mécanisme d'exécution du tronçon et le processus de constitution de sa fenêtre cible $w_t$ qui en découle. Nous rappelons qu'un tronçon est une partie d'un chemin séquentiel, c'est donc une composition de fonctions. L'application d'un tronçon $t = f_k...o...f_2 o f_1$ à une donnée $d$ se traduit par l'application successive des fonctions $f_1, f_2, ...$ et $f_k$ à la donnée $d$. A l'exception de la fonction $f_1$, chaque fonction du tronçon s'applique au résultat produit par l'application de la fonction précédente. Cela signifie que si $f_1 : d_1 \longrightarrow r_1$, $f_2 : d_2 \longrightarrow r_2$, ..., $f_k : d_k \longrightarrow r_k$ désignent respectivement les applications des fonctions $(f_i)_{i=1..k}$ à leurs données respectives $(d_i)_{i=1..k}$ et $(r_i)_{i=1..k}$ représentent leurs résultats respectifs alors :

$$d_1 = d,\ d_2 = r_1,\ d_3 = r_2,\ ...,\ d_k = r_{k-1} \text{ et } r_k = r.$$

Maintenant, considérons pour chaque fonction du tronçon non pas toute sa donnée mais uniquement la partie de celle-ci sur laquelle elle s'applique réellement i.e. sa fenêtre cible. Les applications précédentes deviennent :

$$\tilde{f}_1 : w_{f1} \longrightarrow r_{f1},\ \tilde{f}_2 : w_{f2} \longrightarrow r_{f2},\ ...,\ \tilde{f}_k : w_{fk} \longrightarrow r_{fk}$$

où $(w_{fi})_{i=1..k}$ et $(r_{fi})_{i=1..k}$ désignent respectivement les fenêtres cibles et les fenêtres résultats des fonctions $(fi)_{i=1..k}$. Les fonctions $(\tilde{fi})_{i=1..k}$ représentent les restrictions des fonctions $(fi)_{i=1..k}$ à leurs fenêtres cibles respectives.

**Intuitivement**, on pourrait penser que si $w_t$ est la fenêtre cible associée au tronçon $t$ et $r_t$ sa fenêtre résultat i.e. $\tilde{t} : w_t \longrightarrow r_t$, où $\tilde{t}$ est la restriction de $t$ à sa fenêtre cible, alors on a le système d'équations suivant :

$$w_{f1} = w_t,\ w_{f2} = r_{f1},\ w_{f3} = r_{f2},\ ...,\ w_{fk} = r_{f(k-1)} \text{ et } r_t = r_{fk}.$$

D'après ce système d'équations, la fenêtre cible associée au tronçon $t$ est égale à celle de la fonction au sommet du tronçon i.e. celle de $f1$.

Maleureusement, cette intuition n'est pas toujours confirmée comme nous allons le montrer à travers le contre-exemple suivant. Considérons l'application :

$$cdr\ o\ car\ : << a, b, c >, < d, e, f >> .$$

Pour cet exemple, $t = cdr\ o\ car$. Calculons maintenant la fenêtre cible $w_t$ de $t$.
L'exécution de $t$ se traduit par celle de la fonction *car* puis celle de la fonction *cdr*. La figure 4.8 illustre les règles d'exécution associées à chacune des deux fonctions en considérant la donnée de l'exemple. La partie gauche de chaque règle représente la donnée sur laquelle s'applique la fonction associée ; les nœuds de donnée encerclés constituent la fenêtre cible de la fonction. La partie droite de chaque règle représente le résultat de l'application de la fonction à sa donnée ; les nœuds de donnée encerclés constituent la fenêtre résultat de la fonction. Nous expliquerons dans le paragraphe 4.5.4.2 comment nous avons obtenu ces règles.

Etant données ces règles d'exécution associées aux différentes fonctions du tronçon, déterminons la règle d'exécution $\tilde{t} : w_t \longrightarrow r_t$ associée à tout le tronçon. La partie gauche de cette règle constituera la fenêtre cible que nous recherchons. Nous rappelons que la fenêtre cible associée à un tronçon est le sous-arbre de données réellement utilisé par l'ensemble de ses fonctions lors de son exécution. Dans notre exemple, la fenêtre cible de $t$ contient initialement les nœuds numérotés 1, 2 et 3 (i.e. la fenêtre cible de la fonction *car*). L'exécution de la fonction *car* supprime le nœud 2. Ainsi, le résultat produit i.e. la fenêtre résultat est constitué(e) uniquement des nœuds 1 et 3. Ce résultat n'est pas suffisant pour former la fenêtre cible de la fonction *cdr*

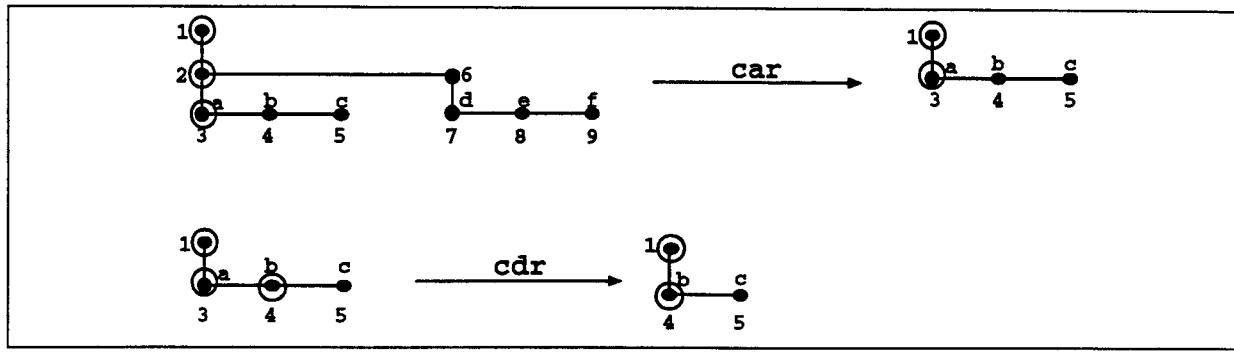



Figure 4.8 : Règles d'exécution des fonctions *car* et *cdr*

car celle-ci utilise le nœud racine ainsi que les deux premiers nœuds de son argument (sa liste). Or, le deuxième élément de la liste n'est pas présent dans la fenêtre résultat produite par la fonction *car*. Il est donc nécessaire de l'ajouter à la fenêtre cible du tronçon $t$. L'application de la fonction *cdr* à sa fenêtre cible ainsi complétée produit sa fenêtre résultat. Cette dernière constitue également la fenêtre résultat du tronçon $t$ puisque la fonction *cdr* est la dernière dans ce tronçon. La figure 4.9 illustre la règle d'exécution $t : w_t \longrightarrow r_t$ obtenue ($w_t = \{1,2,3,4\}$ et $r_t = \{1,4\}$).

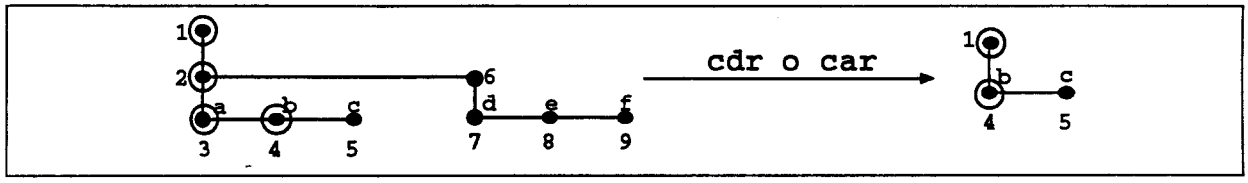



Figure 4.9 : Règle d'exécution du tronçon *cdr o car*

L'exemple ci-dessus montre bien que la fenêtre cible d'un tronçon n'est pas forcément égale à celle de la fonction à son sommet et que la fenêtre résultat de la première fonction (car) est différente de la fenêtre cible (cdr) de la fonction suivante. Un outil de *calcul automatique de cette fenêtre* doit être défini. De plus, un tel outil doit disposer des règles d'exécution (fenêtres cibles+fenêtres résultats) de toutes les fonctions. Il est donc nécessaire d'associer à chaque fonction du langage utilisé une règle d'exécution.
Par ailleurs, dans l'exemple présenté la fenêtre cible calculée i.e. $\{1,2,3,4\}$ est décrite par une simple énumération de nœuds de données. Cette fenêtre se traduit par "prendre le nœud racine de la donnée initiale, puis la racine de son premier argument qui est une liste et enfin les deux premiers éléments de cette liste (le premier fils et le frère de celui-ci)". Dans ce cas précis, le nombre de nœuds de la fenêtre est connu et n'est pas important. De ce fait, leur énumération ne pose aucun problème. Mais imaginons, par exemple, le cas où le tronçon contient la fonction *reverse* appliquée à une longue liste dont la taille ne peut être connue qu'à l'exécution. Deux problèmes sont alors posés. Premièrement, la gestion du descripteur est complexe en termes d'espace de stockage et de temps de traitement. Deuxièmement, les nœuds de données

sont constitués dynamiquement. Par conséquent, leur énumération est impossible. Il est donc indispensable de trouver *un outil de description des fenêtres.*

Pour résumer, nous avons besoin d'un outil de calcul automatique de la fenêtre cible d'un tronçon et d'un outil de description de celle-ci. A travers l'exemple ci-dessus, nous avons illustré le procédé de constitution de la fenêtre cible d'un tronçon. Ce procédé est guidé par le mécanisme d'exécution du tronçon. Notre but est de prévoir un tel calcul par ***analyse statique*** (à la compilation) de chaque tronçon du programme. Pour ce faire, nous avons utilisé une approche basée sur une ***exécution abstraite.*** De plus, dans cette approche, les phrases "prendre le fils", "prendre le frère", ..., utilisées ci-dessus pour la description de la fenêtre, sont exprimées par des lettres. Finalement, une fenêtre est décrite par un mot appartenant à un langage de description d'arbres. L'alphabet et les règles de production d'un tel langage doivent être définis.
Par ailleurs, nous avons vu que le calcul de la fenêtre cible d'un tronçon nécessitait la connaissance de la règle d'exécution associée à chacune de ses fonctions. Pour effectuer un calcul abstrait de cette fenêtre, il est donc nécessaire d'associer une règle appelée **règle d'exécution abstraite ou *REA*** à chaque fonction du langage utilisé. Une telle règle est liée à la sémantique d'exécution de la fonction. Elle se présente sous la forme : $\tilde{f} : Dw_f \rightarrow Dr_f$ où $Dw_f$ et $Dr_f$ désignent les descripteurs associés respectivement à la fenêtre cible et la fenêtre résultat de la fonction $f$. Les règles d'exécution abstraite associées aux fonctions représentent les règles de production du langage de description de fenêtres.

La suite de cette section est organisée comme suit. Avant tout, nous donnons l'alphabet utilisé pour la description des fenêtres. Ensuite, nous présentons les règles d'exécution abstraite associées à chaque fonction du langage utilisé, en l'occurrence le langage Graal. Enfin, nous décrivons l'algorithme utilisé pour le calcul du descripteur associé à la fenêtre d'un tronçon donné.

### 4.5.2 Alphabet de description

L'approche de description utilisée est liée à la représentation de la donnée. Dans le modèle $P^3$, la structure choisie pour représenter une donnée est un arbre binaire. Une fenêtre, étant un sous-arbre de donnée, peut être vue comme un ensemble de chemins reliant entre eux des nœuds de donnée. Par conséquent, la description d'une fenêtre se traduit par la description de tous les chemins la constituant. Par ailleurs, un chemin est vu comme une suite de déplacements de deux types : "aller vers le bas" (prendre le fils) et "aller à droite" (prendre le frère). Si nous codons ces derniers respectivement par les lettres **b** (pour **bas**) et **d** (pour **droite**) alors un chemin (et donc une fenêtre) peut être décrit(e) par une concaténation de lettres (un mot) appartenant à l'ensemble $\{b, d\}$. Cet ensemble, que nous appellerons $\Sigma = \{b, d\}$, représente l'alphabet de description de fenêtres.

L'alphabet $\Sigma$ et l'opérateur de concaténation (et donc de puissance) permettent de décrire un chemin d'une arbre de données. Par exemple, considérons l'exemple de la figure 4.10. Cet exemple représente la fenêtre cible de la fonction *append.* Le chemin $< 0, 1, 2, 3, ..., n+1 >$ est codé par : $bbd^* \equiv b^2d^*$
L'autre chemin de la fenêtre i.e. $< 0, 1, n+2, n+3 >$ est codé par "*bdb*".

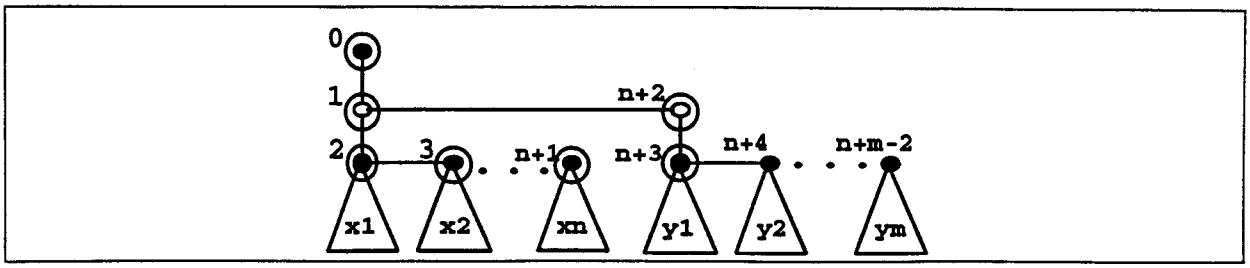



Figure 4.10 : Fenêtre cible associée à la fonction *append*

Par ailleurs, pour pouvoir coder toute la fenêtre cible associée à la fonction *append* il est nécessaire d'avoir un opérateur qui permette d'exprimer l'union de deux chemins. Pour ce faire, nous utiliserons le symbôle "+" d'union classique. Ainsi, la fenêtre associée à la fonction *append* est codée par le descripteur suivant :

$$b^2d^* + bdb$$

**Remarque 1** : Une fenêtre n'est jamais vide. Une fenêtre ne contenant qu'un seul nœud de donnée est décrite par le mot vide $\epsilon$. Le mot vide représente l'élément neutre de l'union.

**Remarque 2** : L'opération d'union est commutative et associative.

**Remarque 3** : L'opération de concaténation n'est pas commutative. Elle est associative et distributive par rapport à l'union. Le descripteur de l'exemple ci-dessus peut s'écrire :

$$b^2d^* + bdb \equiv b(bd^* + db)$$

### 4.5.3 Règle d'exécution abstraite d'une fonction

Dans ce paragraphe, nous allons donner une formalisation de la notion de règle d'exécution abstraite d'une fonction i.e. des notions de fenêtre cible et de fenêtre résultat d'une fonction.

Soit f une fonction, $w_f$ sa fenêtre cible et $r_f$ sa fenêtre résultat. La REA associée à la fonction f est :

$$\tilde{f} \ : \ w_f \longrightarrow r_f = f(w_f)$$

Par abus de langage, nous ne distinguerons pas dans la suite la fonction f et sa restriction $\tilde{f}$ à un sous-arbre de même racine que son argument.

**Définition 4.17**: Les fenêtres $w_f$ et $r_f$ peuvent être définies formellement de la façon suivante : $\forall\ u, v \in \Sigma^*$ :

- $f(w_f u) = f(w_f)u = r_f u$
- $f(w_f+v) = \begin{cases} r_f & si\ \exists (a \neq \epsilon), w, v' \in \Sigma^* \ : w_f = aw \ et \ v = av' \ et \ f(a) \neq a \\ r_f + v & sinon. \end{cases}$

**Proposition 4.1**

Si $v = w_f v'$ alors $f(w_f + v) = r_f v'$.

**Preuve**

$$f(w_f + v) = f(w_f + w_f v')$$

$$= f(w_f(\epsilon + v'))$$
$$= f(w_f v')$$
$$f(w_f + v) = r_f v'$$

Examinons notre formalisation sur les deux fonctions "cdr" et "cons".

### 1) Cas de la fonction "cdr"

La REA associée à la fonction "cdr" est : $\tilde{cdr} : bd \longrightarrow b$.

**a) Propriété 1**

D'après la définition de la fonction "cdr" (figure 4.11.(a)), nous avons :

$$\forall\, u \in \Sigma^* : \text{cdr(bdu)} = \text{bu}$$

Autrement dit, $cdr(w_{cdr}u) = r_{cdr}u$

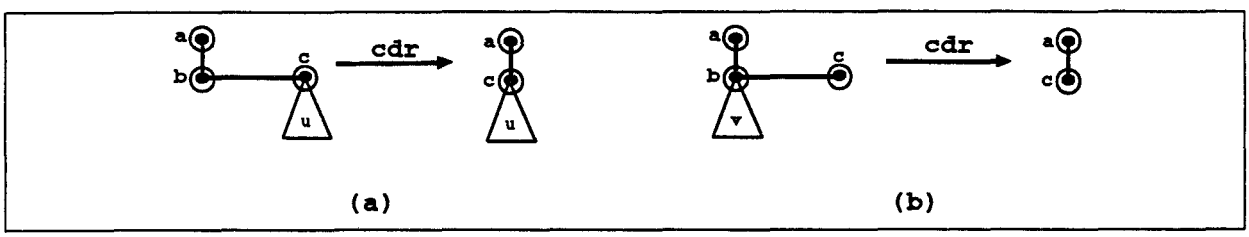



Figure 4.11 : Vérification des propriétés 1 et 2 sur la fonction "cdr"

**b) Propriété 2**

Soient $u, v \in \Sigma^*$ tels que : v = bu. Ajoutons le chemin v à la fenêtre de "cdr" et appliquons la fonction "cdr" à l'arbre obtenu.

D'après la définition de la fonction "cdr" (figure 4.11.(b)), nous avons :

$$\text{cdr(bd + v)} = \text{cdr(bd + bu)} = \text{b}$$

Autrement dit, $cdr(w_{cdr} + v) = r_{cdr}$.

La partie préfixe commune entre la fenêtre cible $w_{cdr}$ et le chemin v ajouté est le chemin "b". Par ailleurs, l'application de la fonction "cdr" à sa fenêtre cible a un effet sur cette partie commune. En effet, celle-ci est supprimée. On est donc dans le premier cas de la propriété 2 de la définition 4.17, ce qui justifie le résultat.

### 2) Cas de la fonction "cons"

La REA associée à la fonction "cons" est : $\tilde{cons} : bdb \longrightarrow bd$.

**a) Propriété 1**

D'après la définition de la fonction "cons" (figure 4.12.(a)), nous avons :

$$\forall\, u \in \Sigma^* : \text{cons(bdbu)} = \text{bdu}$$

Autrement dit, $cons(w_{cons}u) = r_{cons}u$.

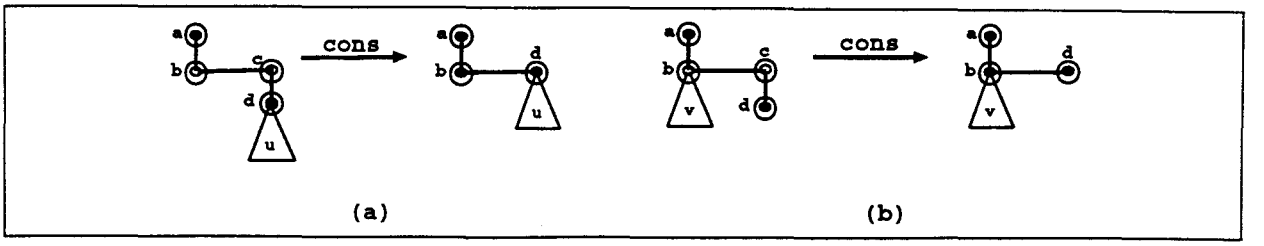



Figure 4.12 : Vérification des propriétés 1 et 2 sur la fonction "cons"

**b) Propriété 2**
Soient $u, v \in \Sigma^*$ tels que : v = bu. Ajoutons le chemin v à la fenêtre de "cons" et appliquons la fonction cons à l'arbre obtenu.
D'après la définition de la fonction "cons" (figure 4.12.(b)), nous avons :

$$cons(bdb + v) = cons(bdb + bu) = bd + bu = bd + v$$

Autrement dit, $cons(w_{cons} + v) = r_{cons} + v$
La partie préfixe commune entre la fenêtre cible $w_{cons}$ et le chemin v ajouté est le chemin "b". Par ailleurs, l'application de la fonction "cons" à sa fenêtre cible n'a aucun effet sur cette partie commune. On est donc dans le deuxième cas de la propriété 2 de la définition 4.17, ce qui justifie le résultat.

### 4.5.4 Règles d'exécution abstraite des fonctions Graal

Dans ce paragraphe, nous donnons le principe de construction des règles d'exécution abstraite associées aux fonctions du langage Graal ainsi que les règles elles-mêmes. Rappelons que ces dernières peuvent être des fonctions primitives, des formes fonctionnelles ou des définitions.

#### 4.5.4.1 Principe de construction

La construction des règles d'exécution abstraite associées aux fonctions du langage Graal est liée, comme il a été indiqué dans le paragraphe précédent, à la représentation des données dans le modèle. La construction des règles d'exécution (fenêtres cibles + fenêtres résultats) obéit aux conditions suivantes :

**Condition 1** : Le nœud racine de l'argument (ou nœud cible) d'une fonction fait toujours partie de la fenêtre cible de celle-ci. Ce nœud représente le "point d'accès" à l'argument de la fonction.

**Condition 2** : Le nœud racine de l'argument d'une fonction doit également être racine de la fenêtre résultat de l'exécution de cette fonction même si celle-ci implique une suppression de nœuds de donnée. Ceci est important lors de l'exploitation du parallélisme vertical.

**Remarque** : L'application d'une règle d'exécution à une donnée doit conserver les liens de type "prédécesseur" i.e. les liens de type "fils-père" et ceux de type "frère droit-frère gauche". Le but est de préparer la donnée non référencée aux opérations de ramasse-miettes.

**Exemple 1 : la fonction *car***
La fonction *car*, appliquée à son argument (qui est une liste), retourne le premier élément de la

liste. Selon la condition 2, le nœud racine de la fenêtre résultat de la fonction *car* ne doit pas être le nœud 1 mais le nœud 0 (figure 4.13). Les informations contenues dans le nœud 1 doivent alors être copiées dans le nœud 0. D'après la remarque ci-dessus, le nœud 2 doit faire partie de la fenêtre cible de la fonction. Par conséquent, l'exécution de la fonction *car* a besoin des trois nœuds entourés i.e. 0, 1 et 2. Le chemin reliant ces trois nœuds constitue alors la fenêtre cible de la fonction *car*. La fenêtre résultat de cette fonction est le chemin reliant les deux nœuds 0 et 2. La *REA* associée à la fonction *car* est donc la suivante : $b^2 \longrightarrow b$.

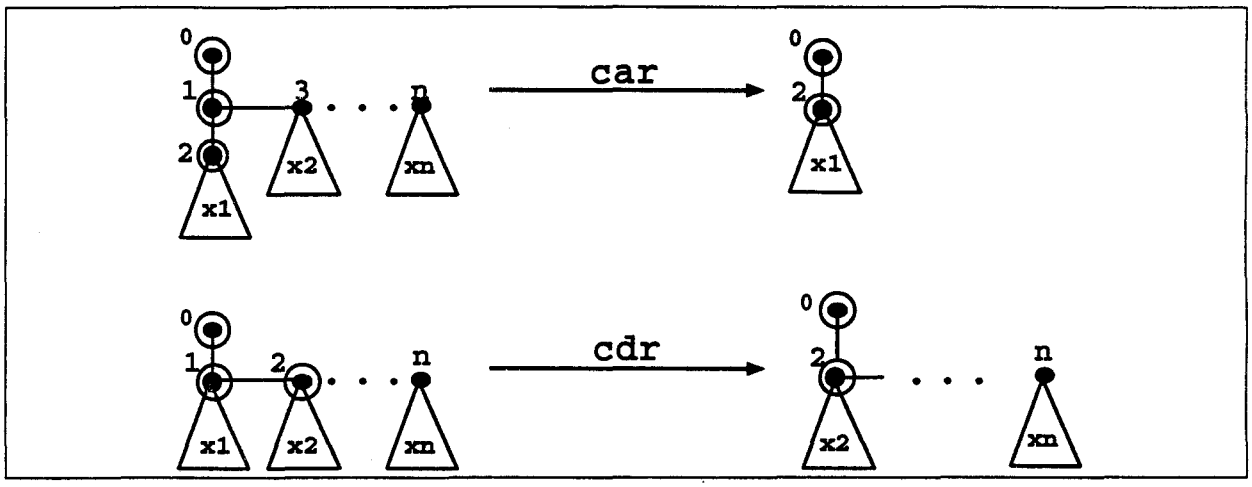



Figure 4.13 : Règles associées aux fonctions *car* et *cdr*

**Exemple 2 : la fonction *cdr***
L'application de la fonction *cdr* à son argument, qui est une liste, retourne la queue de la liste i.e. $< 2, 3, ..., n >$ sur la figure 4.13. Le nœud 0 doit alors désigner le nœud 2. Mais pour pouvoir accéder à ce dernier on a besoin du nœud 1. Par ailleurs, d'après la remarque précédente, on a besoin du nœud 2. La fenêtre cible de la fonction *cdr* est donc constituée des nœuds 0, 1 et 2. A l'issue de l'exécution de la fonction, le nœud 1 est supprimé. La fenêtre résultat comprend, de ce fait, les deux nœuds 0 et 2. Ainsi, la *REA* associée à la fonction *cdr* est : bd⟶b.

Les sections suivantes présentent les règles d'exécution abstraite associées aux différentes fonctions du langage Graal.

### 4.5.4.2 Les fonctions primitives

Le tableau 4.1 montre les règles d'exécution abstraite associées aux différentes fonctions primitives. Ces règles d'exécution ont été obtenues en respectant les règles de construction du paragraphe précédent. Ces règles sont illustrées par les figures 4.14 et 4.15. Dans le tableau, la puissance $i$ utilisée dans la partie droite de la règle associée à la fonction *append* désigne la longueur de la liste premier argument de cette fonction.

| Fonction | Règle associée |
|---|---|
| *car* | $b^2 \longrightarrow b$ |
| *cdr* | $bd \longrightarrow b$ |
| *reverse* | $bd^* \longrightarrow bd^*$ |
| *length* | $\epsilon \longrightarrow \epsilon$ |
| *list* | $\epsilon \longrightarrow \epsilon$ |
| *append* | $b^2d^l + bdb \longrightarrow bd^{l+1}$ |
| *cons* | $bdb \longrightarrow bd$ |
| $S_i$ | $bd^ib \longrightarrow b$ |
| *Id* | $\epsilon \longrightarrow \epsilon$ |
| *+,-,*,/,div,mod* | $bd^* \longrightarrow \epsilon$ |
| *Null, test de type, ...* | $\epsilon \longrightarrow \epsilon$ |

Tableau 4.1 : Règles associées aux fonctions primitives de Graal

#### 4.5.4.3 Les formes fonctionnelles

Les formes fonctionnelles prédéfinies de Graal sont : les formes fonctionnelles de *distribution*, la forme fonctionnelle *composition*, les formes fonctionnelles *binaires-unaires*, les formes fonctionnelles conditionnelles *if*, *Cond* et *While*, la forme fonctionnelle *constante* et la forme fonctionnelle "*::*". Dans la suite de ce paragraphe, nous donnerons les règles d'exécution abstraite associées à toutes ces formes fonctionnelles.

**Les formes fonctionnelles de distribution :** Nous rappelons qu'il existe trois formes fonctionnelles de distribution : *distl*, *distr* et *dista*. D'après leurs définitions, ces formes fonctionnelles peuvent s'exprimer chacune comme une composition de fonctions du langage *FP*, ce qui permet d'introduire des opérations plus basiques. En effet, nous avons les équivalences suivantes :

- $(distl\ f)_{Graal} \equiv (\alpha f\ o\ distl)_{FP}$
- $(distr\ f)_{Graal} \equiv (\alpha f\ o\ distr)_{FP}$
- $(dista\ f\ g)_{Graal} \equiv (f\ o\ \alpha g)_{FP}$

où $f$ et $g$ sont deux fonctions quelconques, $\alpha$ est la forme *alpha* du langage $FP$ et *distl* et *distr* sont respectivement les formes fonctionnelles de distribution à gauche et à droite de $FP$.
Il convient alors de remplacer dans le programme fonctionnel (écrit en Graal) chaque fois qu'elles y apparaîssent les trois formes fonctionnelles de distribution par leurs arborescences fonctionnelles équivalentes dans *FP*. Un découpage leur sera alors appliqué. Encore, faut-il associer une *REA* à chacune des trois formes fonctionnelles $\alpha$, *distl* et *distr* de $FP$. Ces règles sont données dans le tableau 4.2. Elles sont également illustrées par la figure 4.16.

**La forme fonctionnelle composition :** De la même façon que pour les formes fonctionnelles de distribution, la forme fonctionnelle *composition* peut s'exprimer comme une composée de deux

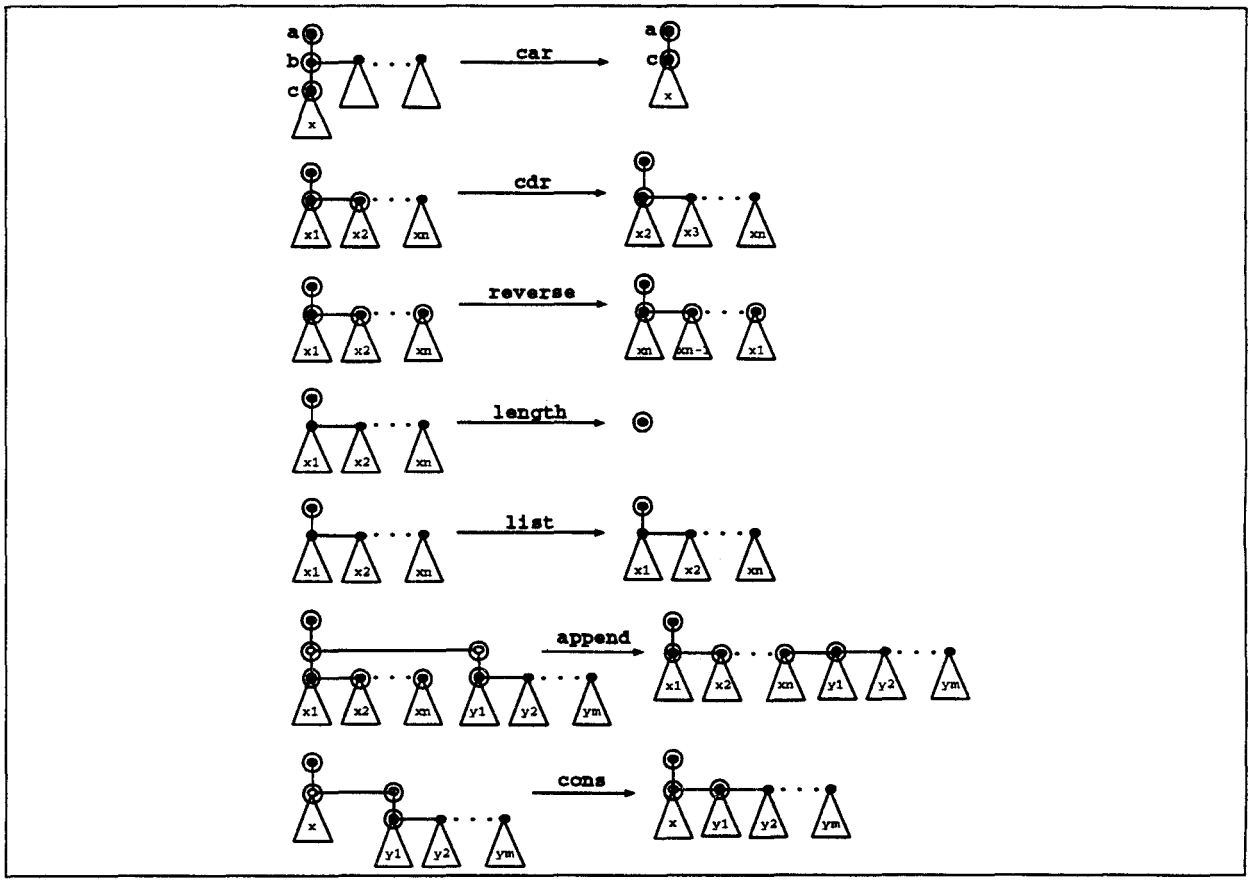



Figure 4.14 : Règles associées aux fonctions de manipulation de listes

| Fonction | Règle associée |
|---|---|
| $(\alpha)_{FP}$ | $bd^* \longrightarrow bd^*$ |
| $(distl)_{FP}$ | $bdbd^* \longrightarrow bd^*bd$ |
| $(distr)_{FP}$ | $b^2d^* + bd \longrightarrow bd^*bd$ |

Tableau 4.2 : Règles associées aux fonctions $(\alpha)_{FP}$, *distl* et *distr* de $FP$

fonctions du langage $FP$. L'expression équivalente est la suivante :

$$(\{f\ g_1\ g_2 \cdots g_n\})_{Graal} \equiv (f\ o\ [g_1\ g_2 \cdots g_n])_{FP}$$

Une *REA* doit alors être associée à la fonction *construction* notée *[]* de $FP$. Cette règle est la même que celle de la fonction alpha de $FP$ pour les mêmes raisons. On a donc :

$$([])_{FP} : \epsilon \longrightarrow \epsilon$$

**Les formes conditionnelles if, Cond et While :** Les formes fonctionnelles conditionnelles sont des structures de contrôle. Avec leurs paramètres, elles forment des arborescences fonction-

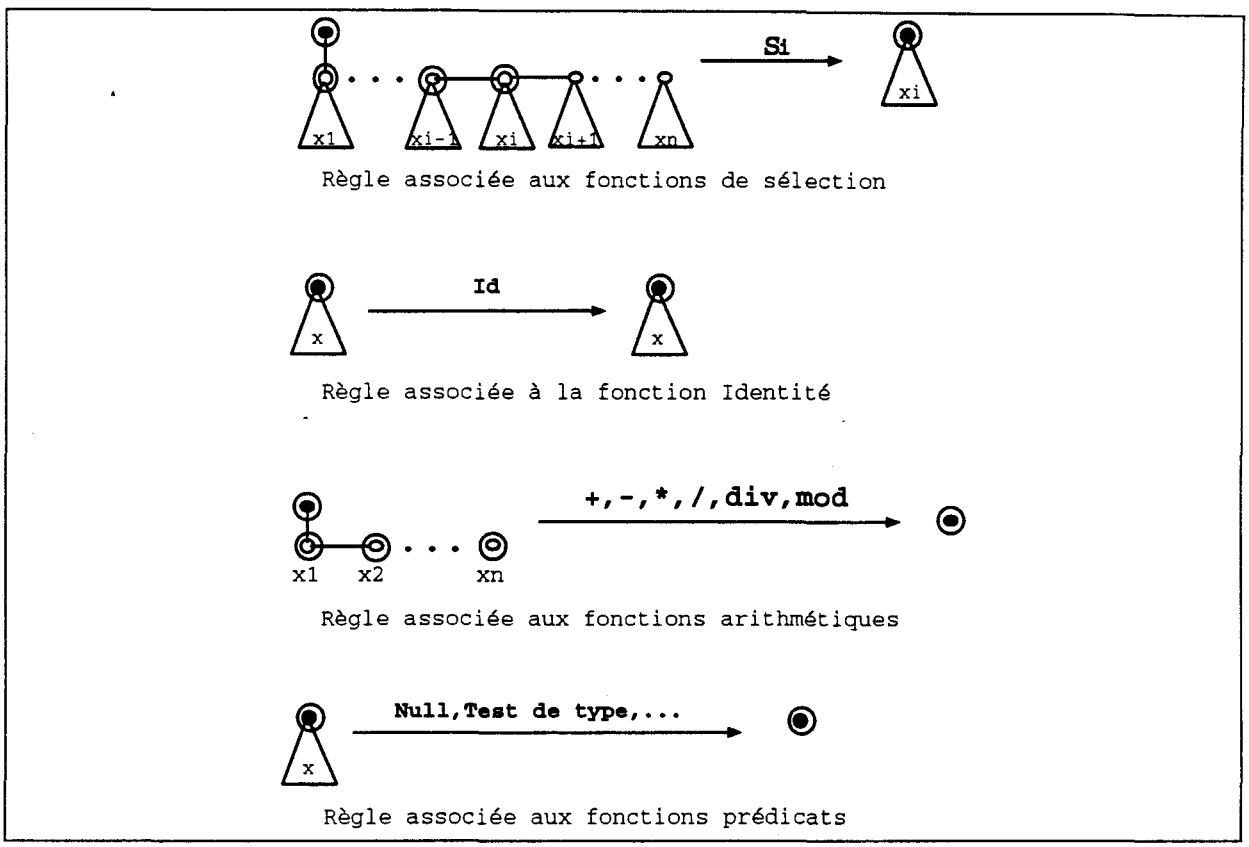



Figure 4.15 : Règles associées aux autres fonctions primitives

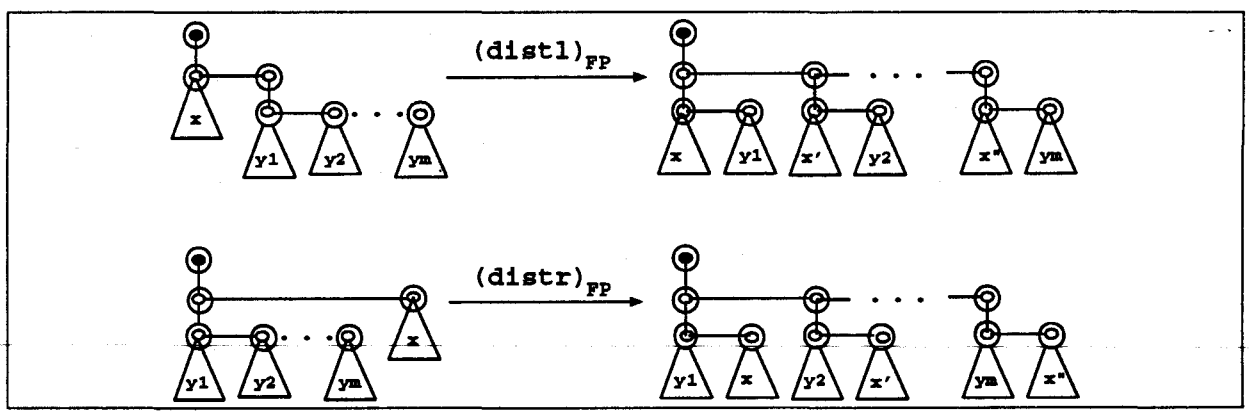



Figure 4.16 : Illustration des règles associées aux fonctions *distl* et *distr* de $FP$

nelles qui doivent être découpées en tronçons. La *REA* qui leur est associée est la suivante :

$$if, Cond, While : \epsilon \longrightarrow \epsilon$$

**La forme fonctionnelle constante :** La forme fonctionnelle *constante*, notée "\", remplace l'argument courant (qui devient son argument) par son objet paramètre. Son exécution nécessite donc la connaissance de la racine de l'argument. Par conséquent, sa *REA* est la suivante :

$$\backslash \; : \; \epsilon \longrightarrow \epsilon$$

**La forme fonctionnelle "::" :** La forme fonctionnelle "::" s'exprime également comme une composée de fonctions de $FP$ de la façon suivante :

$$(f \; :: \; a_1 \; a_2 \; ... \; a_n)_{Graal} \equiv (\{f \; \backslash a_1 \; \backslash a_2 \; ... \; \backslash a_n\})_{Graal} \equiv (f \; o \; [\backslash a_1 \; \backslash a_2 \; ... \; \backslash a_n])_{FP}$$

Cette forme fonctionnelle doit donc être remplacée dans le programme fonctionnel par son arborescence fonctionnelle équivalente en $FP$. C'est pourquoi il n'est pas nécessaire d'associer une *REA* à cette fonction.

**Les formes fonctionnelles binaires-unaires :** Il existe deux types de formes fonctionnelles binaires-unaires : la forme *binu* et la forme *binul*. Comme le montre la figure 4.17, ces deux fonctions construisent une séquence de deux arguments à partir de leur paramètre x et leur argument y. La *REA* associée à chacune des deux formes fonctionnelles est la suivante :

$$binu, binul \; : \; \epsilon \longrightarrow bd$$

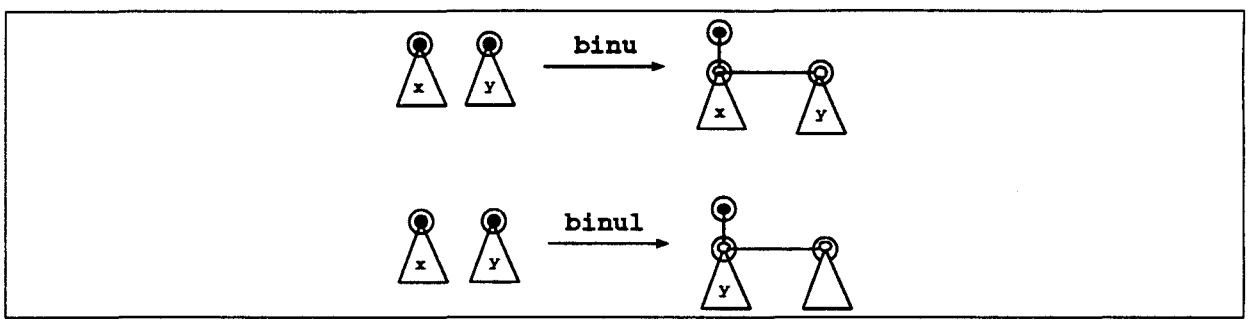



Figure 4.17 : Illustration des règles associées aux fonctions binaires-unaires

#### 4.5.4.4 Les définitions

Une définition comporte une ou plusieurs arborescences fonctionnelles. Sa *REA* associée est telle que sa fenêtre cible est égale à celle du tronçon racine de son arborescence principale. Sa fenêtre résultat est égale à celle du dernier tronçon exécuté lors de l'évaluation de la dernière arborescence de cette définition.
Par ailleurs, la règle associée à l'appel d'une définition dans un programme fonctionnel est la suivante :

$$\epsilon \longrightarrow \epsilon$$

Ceci parce que l'appel de la définition se traduit par le passage du nœud de donnée racine de l'argument au tronçon racine de son arborescence principale.

La section suivante présente la méthode de calcul de la *REA* associée à une composée de fonctions.

### 4.5.5 Règle d'exécution abstraite associée à une composée de fonctions

#### 4.5.5.1 Calcul de la règle

**Définition 4.18** : Etant donnés deux chemins codés respectivement par u et v. Le chemin u est dit **préfixe partiel** de v si et seulement s'il existe un chemin $(v' \neq \epsilon) \in \sum^*$ tel que : v=uv'.

**Remarque** : $\epsilon$ est préfixe partiel de n'importe quel mot.

**Définition 4.19** : Soit v un descripteur associé à une fenêtre $\omega$ donnée. Le mot u est dit **préfixe** de v si et seulement s'il est préfixe partiel d'au moins un chemin de v (i.e. de $\omega$).
Formellement : $\forall\ u, v \in \Sigma^*$, Prefixe(u,v)=VRAI $\Leftrightarrow\ \exists\ u', v' \in \Sigma^*$ : v=uv'+u'.

**Exemples**

1. $u = bd$ et $v = b^2d^* + bdb$
   $v$ est le descripteur associé à la fenêtre cible de la fonction "append". $u$ est préfixe de $v$ car il est préfixe partiel de bdb (v'=b et u'=$b^2d^*$).
2. $u = b$ et $v = bd^i$
   $v$ est le descripteur associé à la fenêtre cible de la fonction "sélecteur". $u$ est préfixe partiel de v et donc préfixe de $v$ (v'=$d^i$ et u'=$\epsilon$).

   **Remarque** : Si $u = bd$ alors u est préfixe de v seulement si $i \geq 2$.

A partir de cette définition, on peut définir la fonction booléenne *Prefixe* par :

$$Prefixe(u, v) = \begin{cases} VRAI & si\ u\ est\ prefixe\ de\ v \\ FAUX & sinon. \end{cases}$$

Cette fonction est utilisée pour l'implantation de l'algorithme de découpage d'un programme en segments.

Le problème du calcul de la *REA* associée à une composée de deux fonctions est posé de la façon suivante :

**Le problème**

Etant données les *REA* ci-dessous associées à deux fonctions f et g.

$$\tilde{f} : w_f \rightarrow r_f$$
$$\tilde{g} : w_g \rightarrow r_g$$

Déterminer la *REA* associée à la fonction $gof$.

$$\tilde{gof} : w_{gof} \rightarrow r_{gof}$$

Notre algorithme de calcul de la *REA* associée à une composée de deux fonctions est basé sur les différents cas étudiés ci-dessus.

#### 4.5.5.2 Etude de cas et exemples

**1) Cas où $w_g$ est préfixe de $r_f$**

$Prefixe(w_g, r_f) = VRAI \Leftrightarrow \exists\, r(r \neq \epsilon), u \in \Sigma^* / \; r_f = w_g r + u$

- $w_{gof} = w_f$ (4.1)
- $r_{gof} = \begin{cases} r_g r & si\ \exists (a \neq \epsilon), w, u' \in \Sigma^* : w_g = aw\ et\ u = au'\ et\ g(a) \neq a \\ r_g r + u & sinon. \end{cases}$ (4.2)

**Preuve**

1. Montrons que : $w_{gof} = w_f$
   La fenêtre cible associée à la fonction $gof$ est composée de l'ensemble des noeuds de données dont ont effectivement besoin les fonctions f et g pour s'exécuter. Par conséquent, pour calculer $w_{gof}$ on considère dans un premier temps les noeuds de $w_f$. Ensuite, on complète avec les noeuds dont a besoin la fonction g et qui ne sont pas dans la fenêtre $r_f$. Or dans notre cas, tous les noeuds nécessaires à l'exécution de g sont dans $r_f$. Autrement dit, $w_f$ suffit pour exécuter la fonction $gof$ c'est à dire : $w_{gof} = w_f$.

2. Calculons $r_{gof}$
   $r_{gof} = gof(w_{gof})$
   $= gof(w_f)$ d'après 4.1
   $= g(f(w_f))$
   $= g(r_f)$
   $= g(w_g r + u)$ par hypothèse
   $r_{gof} = \begin{cases} g(w_g r) = r_g r & si\ \exists (a \neq \epsilon), w, u' \in \Sigma^* : w_g = aw\ et\ u = au'\ et\ g(a) \neq a \\ g(w_g) r + u = r_g r + u & sinon. \end{cases}$

**Exemple**

Supposons que : $f = (disl)_{FP}$ et $g = S_i$. Rappelons que les $REA$ associées à ces deux fonctions sont :
- $(\tilde{disl})_{FP} : bdbd^*b \rightarrow bd^*bd$
- $\tilde{S_i} : bd^ib \rightarrow b$.

Dans cet exemple, on a : $r = d$, $u = b(\epsilon + d + ... + d^{i-1} + d^{i+1} + ...)bd$ et $a = b$. Par conséquent, la $REA$ associée à la fonction $(S_i\ o\ (disl)_{FP})$ est la suivante :

$$S_i\ o\ (disl)_{FP} : bdbd^*b \rightarrow bd$$

**2) Cas où $w_g = r_f$**

- $w_{gof} = w_f$ (4.3)
- $r_{gof} = r_g$ (4.4)

**Preuve**

1. Montrons que : $w_{gof} = w_f$
   Pour les mêmes raisons que dans le cas précédent, on a : $w_{gof} = w_f$.

2. Montrons que : $r_{gof} = r_g$
   $r_{gof} = gof(w_{gof})$
   $= gof(w_f)$ d'après (4.3)
   $= g(f(w_f))$
   $= g(r_f)$
   $= g(w_g)$ par hypothèse
   $r_{gof} = r_g$

**Exemple**

Cette situation peut être illustrée par : $f = sub$ et $g = null$. Les *REA* associées à ces deux fonctions sont :
- $\tilde{sub} : bd^* \rightarrow \epsilon$
- $\tilde{null} : \epsilon \rightarrow \epsilon$.

D'après les règles ci-dessus et les équations 4.3 et 4.4, la *REA* associée à la fonction composée (*car o cdr*) est :

$$\tilde{null\ o\ sub} : bd^* \rightarrow \epsilon$$

**3) Cas où $r_f$ est préfixe de $w_g$**
$Prefixe(r_f, w_g) = VRAI \Leftrightarrow \exists\ w\ (w \neq \epsilon), v \in \Sigma^*\ /\ w_g = r_f w + u$

- $w_{gof} = w_f w + u$ (4.5)
- $r_{gof} = r_g$ (4.6)

**Preuve**

**Proposition 4.2**
L'application d'une fonction quelconque à sa fenêtre est injective i.e. $\forall u \in \Sigma^*,\ f(w_f) = f(u) \Rightarrow w_f = u$

**Preuve**
Supposons que : $\exists u \in \Sigma^* : f(w_f) = f(u)\ et\ w_f \neq u$
On a : $w_f \neq u \Rightarrow \exists v \in \Sigma^*, u = w_f v\ ou\ u = w_f + v$

i) Supposons que : $\exists v \in \Sigma^*, u = w_f v$
On a : $f(u) = f(w_f v) = f(w_f)v = f(u)v \Rightarrow v = \epsilon$
D'où : $w_f = u$

ii) Supposons que : $\exists v \in \Sigma^*, u = w_f + v$
On a : $f(u) = f(w_f + v) = f(w_f) + v = f(u) + v \Rightarrow v = \epsilon$

D'où : $w_f = u$

1. Montrons que : $r_{gof} = r_g$
   La fenêtre résultat de la fonction $gof$ est l'ensemble des noeuds de données produits après application de cette fonction à sa fenêtre cible. On peut dire également que c'est l'union de l'ensemble des noeuds de $r_f$ n'appartenant pas à $w_g$ et de l'ensemble des noeuds de $r_g$. Or ici, tous les noeuds de $r_f$ sont dans $w_g$. Par conséquent, $r_{gof} = r_g$.

2. Montrons que : $w_{gof} = w_f w + u$
   $gof(w_{gof}) = r_g$ d'après 4.6
   $= g(w_g)$
   $= g(r_f w + u)$ par hypothèse
   $= g(f(w_f)w + u)$
   $= g(f(w_f w) + u)$ d'après la définition 5.17
   $= g(f(w_f w + u))$ d'après la définition 5.17
   $gof(w_{gof}) = gof(w_f w + u)$ (*)

   (*) et (proposition 5.2) $\Rightarrow w_{gof} = w_f w + u$

**Exemple**

Un exemple représentant une telle situation est celui donné dans le paragraphe 4.5.1 i.e. : $f = car$ et $g = cdr$. Les *REA* associées aux fonctions *car* et *cdr* sont les suivantes :

$$\tilde{car} : b^2 \rightarrow b$$
$$\tilde{cdr} : bd \rightarrow b$$

D'après les règles ci-dessus et les équations 4.5 et 4.6 : $w = d$ et $u = \epsilon$. Par conséquent, la *REA* associée à la fonction composée ($cdr\ o\ car$) est la suivante :

$$cdr\ o\ car : b^2 d \rightarrow b$$

Le calcul de la *REA* associée à un tronçon est une application récursive de cet algorithme aux compositions de toutes les fonctions composant le tronçon.

Dans la section suivante, nous présenterons les motivations et les critères de regroupement des segments (tronçons avec leurs fenêtres) en paquets.

## 4.6 Regroupement de segments en paquets

### 4.6.1 Pourquoi regrouper?

La première étape de transformation du programme consiste à découper celui-ci en tronçons. Ce découpage permet d'identifier le parallélisme exploitable inhérent au programme. L'exploitation de tout le parallélisme potentiel induit un surcoût de communication important à cause du caractère fin de la granularité (c'est une caractéristique connue des langages fonctionnels). Ce qui conduit à une exécution inefficace du programme.

Pour améliorer l'efficacité de l'exécution il faut limiter les communications entre les tronçons du programme. Ceci se traduit par l' élimination du parallélisme de traitement jugé, a priori, inefficace et donc par un regroupement des tronçons en paquets. Le parallélisme est jugé (à la compilation) inefficace lorsque son exploitation ne permet pas, a priori, d'améliorer le temps d'exécution du programme. La figure 4.18 illustre un exemple de situation où l'exploitation du parallélisme horizontal est inefficace. En effet, les tronçons (*car o cdr*) et (*cdr o car*) ne comportent que quelques instructions de modification de liens. Leur exécution prend seulement quelques microsecondes. Par conséquent, leur exécution à distance i.e. sur un autre nœud que celui sur lequel se trouve le code de "[]" ne fera qu'occasionner un coût de communication (quantifié en millisecondes) qu'on ne pourra pas compenser par l'exploitation du parallélisme en question.

Le paragraphe suivant définit les critères utilisés dans notre approche pour le regroupement des segments en paquets.

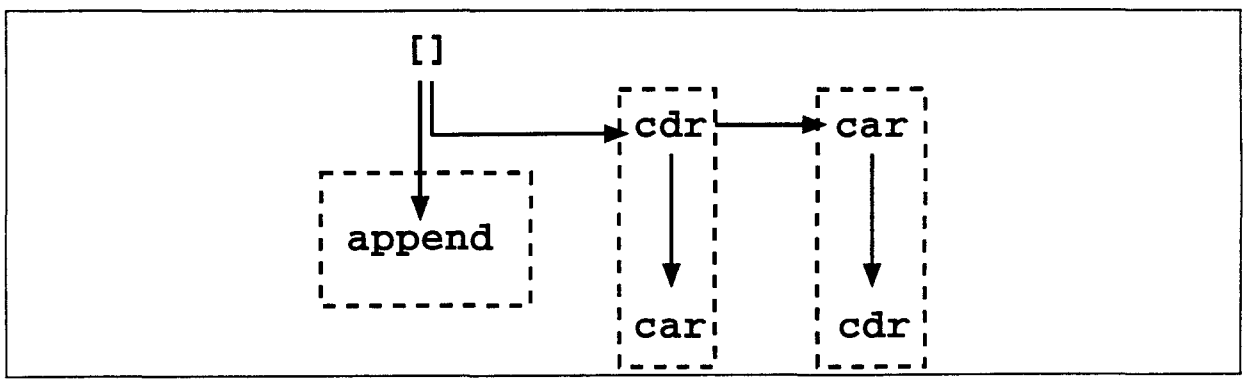



Figure 4.18 : Un exemple de situation où le parallélisme horizontal est inefficace

### 4.6.2 Critères de regroupement

Dans ce paragraphe, nous confondrons parfois le segment avec son tronçon associé.

Avant de donner les critères de regroupement de segments en paquets, introduisons la notion de *tronçon frontière.*

**Définition 4.19 : un tronçon frontière** est un tronçon qui définit le début d'un paquet. Il constitue le point d'entrée de ce dernier : c'est le premier tronçon exécuté lors de l'exécution du paquet.

Le regroupement des segments en paquets consiste en l'identification de tous les tronçons frontières dans le programme. De ce fait, les critères de regroupement sont les critères selon lesquels un tronçon du programme est un tronçon frontière. Par ailleurs, le parallélisme est

identifié dans le programme à l'aide de fonctions frontières. Par conséquent, la limitation du parallélisme à exploiter se traduit par une renonciation à certaines de ces fonctions frontières. Les critères de regroupement des segments en paquets sont donc liés au type de fonction frontière qui définit chaque tronçon du programme. Nous rappelons qu'il existe quatre types de fonction frontière (cf. section 4.4.1). En effet, celle-ci peut être :

- une fonction non stricte ;
- la racine d'un paramètre d'une forme fonctionnelle ;
- une fonction successeur d'une forme fonctionnelle génératrice de parallélisme (ex. $\alpha$, [ ], ...) ;
- une fonction successeur d'une fonction d'ordre supérieur.

Le principe d'un algorithme de regroupement des segments en paquets est de parcourir toutes les arborescences fonctionnelles de segments en commençant par la principale. Le parcours de chaque arborescence commence par le segment racine de celle-ci. Pour chaque tronçon visité, l'algorithme examine d'abord le type de la fonction frontière qui le définit. Ensuite, en fonction de ce type, il décide si le tronçon est un tronçon frontière ou non.

Dans la suite de ce paragraphe, nous allons présenter les critères, basés sur le type de la fonction frontière, utilisés par l'algorithme pour effectuer le regroupement. Dans les figures utilisées pour servir d'exemples illustratifs du mécanisme de regroupement, deux tronçons remplis avec le même motif font partie du même paquet. A l'inverse, deux tronçons remplis avec deux motifs différents appartiennent à deux paquets différents.

1. **Cas d'une fonction non stricte**
   Une fonction non stricte permet d'exprimer le parallélisme vertical (ou d'anticipation). L'efficacité produite par l'exploitation de ce parallélisme dépend du successeur de la fonction non stricte qui l'exprime. En effet, considérons un tronçon t dont la fonction frontière est une fonction non stricte appelée f. Nous avons alors les situations suivantes :

   - La fonction f a un successeur, que nous appellerons g, dans le tronçon t. Si g est une fonction ordinaire stricte (ex. *add* dans la figure 4.19 (a)) alors l'exécution de t est bloquée juste après l'exécution de f. La gestion de ce blocage remet en cause l'efficacité attendue en exploitant l'anticipation. Dans ce cas de figure, le tronçon t ne doit pas être un tronçon frontière. Par contre, si g est une fonction d'ordre supérieur (ex. *apply* dans la figure 4.19 (b)) alors l'anticipation sur l'exécution de t permettra de recouvrir le temps de préparation du code produit par la fonction d'ordre supérieur. Par conséquent, le tronçon t doit être un tronçon frontière i.e. doit commencer un nouveau paquet.
   - La fonction f n'a pas de successeur, elle est unique dans le tronçon t. Dans ce cas, il faudra examiner la position du tronçon t dans le chemin séquentiel où il se trouve. On peut avoir les situations suivantes :
     - Le tronçon t est en fin de chemin séquentiel principal d'une arborescence fonctionnelle (figure 4.20). Le tronçon t doit être un tronçon frontière car ceci permettra d'anticiper l'exécution de la suite du chemin séquentiel.

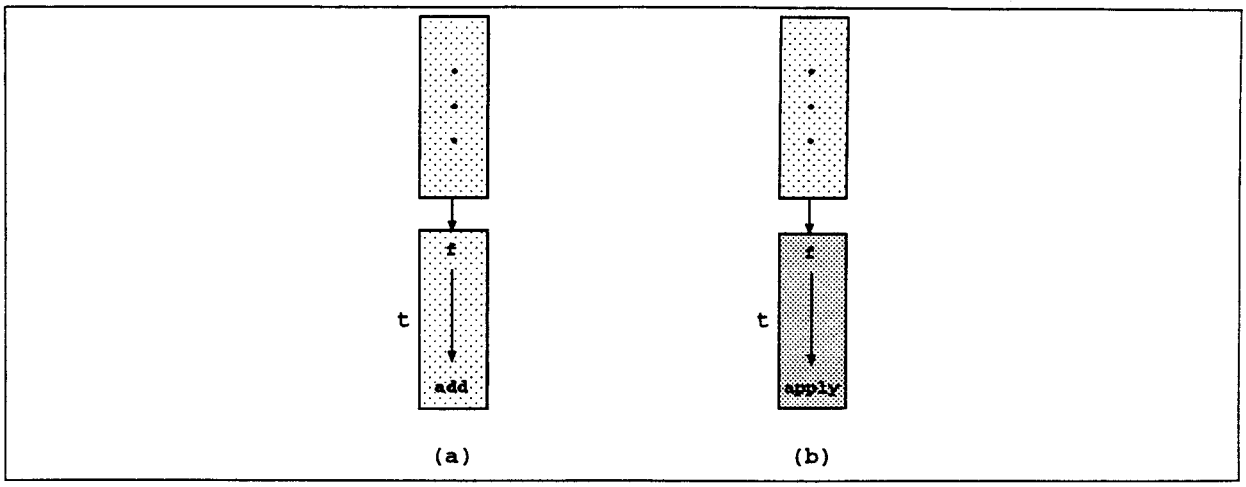



Figure 4.19 : Regroupement : Cas où la fonction non stricte identifiant un tronçon a un successeur dans ce tronçon

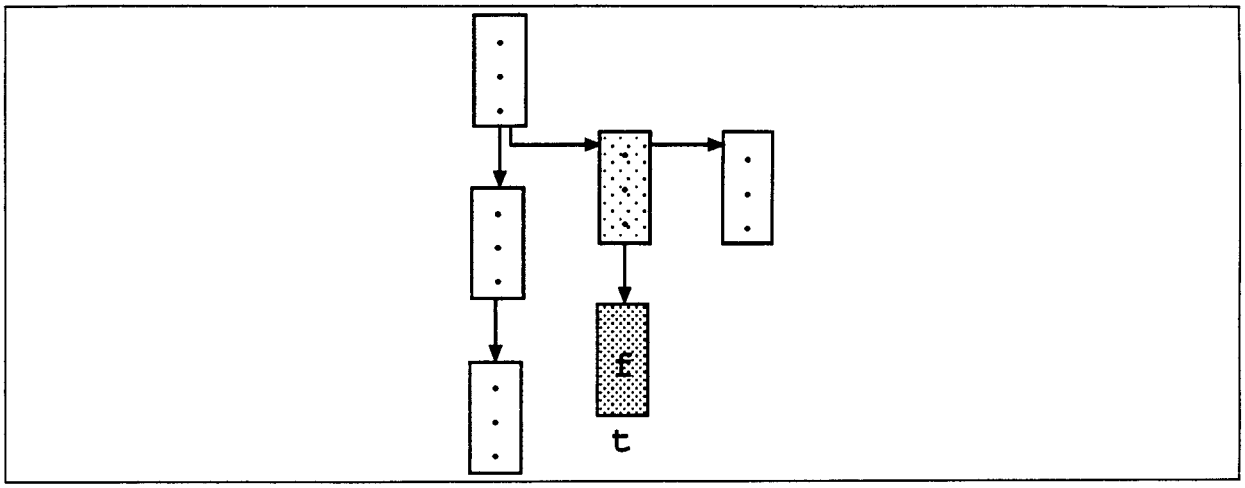



Figure 4.20 : Regroupement : Cas où la fonction non stricte identifiant un tronçon n'a pas de successeur dans ce tronçon et elle est en fin de chemin séquentiel principal d'un paramètre d'une forme fonctionnelle

– Le tronçon t n'est pas en fin d'un quelconque chemin séquentiel (figure 4.21). Dans ces conditions, l'anticipation sur l'exécution de t ne sera pas une source d'efficacité. Par conséquent, le tronçon t ne doit pas être un tronçon frontière.

2. **Cas de la racine d'un paramètre d'une forme fonctionnelle**
Nous rappelons que le parallélisme horizontal de traitement est produit par les formes fonctionnelles et les fonctions polyadiques. De plus, dans le paragraphe 4.5.4.3, nous avons vu que les formes fonctionnelles génératrices de parallélisme peuvent s'exprimer à l'aide des deux formes fonctionnelles (dites de base) du langage $FP$ : la forme *alpha* $((\alpha)_{FP})$ et la forme construction $(([])_{FP})$. Par ailleurs, les fonctions polyadiques comportant un parallélisme de traitement peuvent également être exprimées par les deux formes $([])_{FP}$

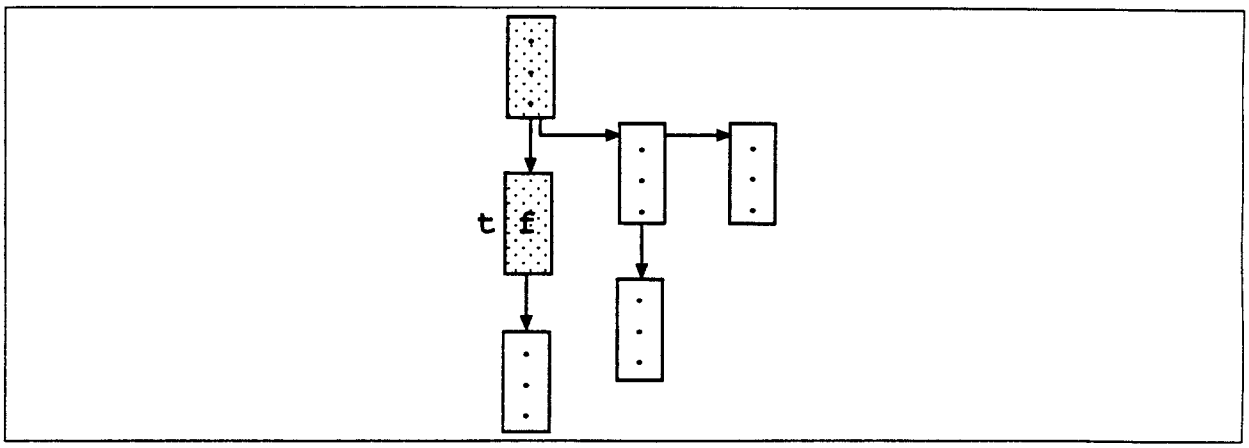



Figure 4.21 : Regroupement : Cas où la fonction non stricte identifiant un tronçon n'a pas de successeur dans ce tronçon et ce dernier n'est en fin d'un quelconque chemin séquentiel

et $(\alpha)_{FP}$. En effet, si f est une fonction polyadique et (f : $P_1$, $P_2$, ..., $P_k$) son application aux fonctions $(P_i)_{i=1,..,k}$ (représentées par des arborescences fonctionnelles) alors cette dernière peut être exprimée par : $f\ o\ ([P_1, P_2, ..., P_k])_{FP}$. On peut remarquer que si les fonctions $(P_i)_{i=1,..,k}$ sont les mêmes i.e. toutes égales à P alors l'application pourra s'écrire : $f\ o\ (\alpha)_{FP}P$.

Il découle de ce qui précède que l'étude du regroupement de segments en présence de formes fonctionnelles génératrices de parallélisme se ramène à l'étude du regroupement en présence des deux formes fonctionnelles $(\alpha)_{FP}$ et $([])_{FP}$ . Par conséquent, nous allons nous limiter à l'examen de ces deux cas.

**a-Cas de la forme fonctionnelle $(\alpha)_{FP}$**

La forme fonctionnelle $(\alpha)_{FP}$ est définie par :

$$(\alpha)_{FP}\ \mathrm{P} : < d_1, d_2, ..., d_k > \equiv < P : d_1, P : d_2, ..., P : d_k >$$

où $< d_1, d_2, ..., d_k >$ est une séquence d'arguments et P est une fonction donnée.
Le regroupement du paramètre P avec la forme $(\alpha)_{FP}$ dépend du découpage appliqué à ce paramètre. En effet, on distingue les situations suivantes :

- Le nombre de paquets résultant du découpage du paramètre P est supérieur à 1. Dans ce cas, le coût de synchronisation des paquets à l'exécution du paramètre est important. De ce fait, cette synchronisation doit être gérée de façon parallèle. Autrement, la forme fonctionnelle $(\alpha)_{FP}$ provoquerait un goulot d'étranglement au sein du processeur qui l'exécuterait. Par conséquent, le paramètre P ne doit pas être regroupé avec la forme $(\alpha)_{FP}$.
- Le nombre de paquets résultant du découpage du paramètre P est égal à 1. Dans ce cas, le regroupement du paramètre P avec sa forme fonctionnelle dépend du découpage en tronçons qui lui a été appliqué. On distingue deux situations possibles :
  - Le nombre de tronçons composant le paquet P est supérieur à 1. Pour les mêmes raisons que le cas présenté ci-dessus, le paramètre P ne doit pas être regroupé

avec sa forme fonctionnelle.

- Le paquet P comporte un seul tronçon. Dans ce cas, P est statiquement regroupé avec sa forme fonctionnelle. Néanmoins, cette décision pourra être remise en cause à l'exécution en fonction de la longueur de la séquence d'arguments à laquelle s'applique la forme fonctionnelle. En ce sens que si la longueur de la séquence dépasse un certain seuil alors on décide dynamiquement de l'éclatement de la forme fonctionnelle regroupée avec son paramètre. Plus de détails sur la façon d'éclater seront donnés dans le chapitre suivant.

**b-Cas de la forme fonctionnelle $([])_{FP}$**
La forme fonctionnelle $([])_{FP}$ est définie par :

$$([P_1, P_2, ..., P_k])_{FP} : d \equiv < P_1 : d, P_2 : d, ..., P_k : d >$$

où $P_1, P_2, ..., P_k$ sont des fonctions (paramètres de la forme fonctionnelle) et d est une donnée.

Le regroupement des paramètres $(P_i)_{i=1,..,k}$ avec la forme $([])_{FP}$ dépend du découpage qui leur a été appliqué. Le même raisonnement appliqué à P dans le traitement de la forme fonctionnelle $(\alpha)_{FP}$ sera utilisé sur chacun de ces paramètres. Dans le cas où un paramètre est composé d'un seul tronçon, il doit être regroupé, de façon statique, avec sa forme fonctionnelle. Un éclatement peut être décidé dynamiquement si la fenêtre associée au paramètre est importante. Plus de précisions sur les conditions et les critères de l'éclatement seront données au chapitre suivant.

3. **Cas d'une fonction successeur d'une forme fonctionnelle génératrice de parallélisme**
Nous rappelons que l'exécution des tronçons regroupés dans un paquet est décidée séquentielle à la compilation. En conséquence, le regroupement d'un tronçon défini par une fonction successeur d'une forme fonctionnelle génératrice de parallélisme dans le même paquet que cette dernière dépend du partitionnement (ou regroupement) appliqué au(x) paramètre(s) de la forme fonctionnelle en question. Selon le résultat du regroupement effectué sur ce(s) paramètre(s), on distingue les situations suivantes :

- Le(s) paramètre(s) de la forme fonctionnelle comporte(nt) du parallélisme. Dans ce cas, le regroupement de la forme fonctionnelle avec le tronçon défini par sa fonction successeur ne respecterait plus la définition d'un paquet. Par conséquent, le tronçon identifié par une fonction successeur d'une forme fonctionnelle génératrice de parallélisme ne doit pas être un tronçon frontière. Dans la figure 4.22(a), le parallélisme inhérent à la forme fonctionnelle *ff* est supposé être exploité à l'exécution puisque les tronçons *t4* et *t5* sont remplis avec des motifs différents. Par conséquent, le tronçon t3 ne doit pas être dans le même paquet que celui qui contient t2, c'est à dire que t3 est tronçon frontière. Ce qui justifie la différence de motifs de t2 et t3 sur la figure.
- Le(s) paramètre(s) de la forme fonctionnelle ne comporte(nt) pas de parallélisme i.e. il(s) est(sont tous) regroupé(s) dans le même paquet que celui qui contient la forme fonctionnelle. Dans ces conditions, cette dernière est considérée comme une fonction simple ne comportant pas de parallélisme. De ce fait, la décision de considérer le tronçon identifié par sa fonction successeur comme tronçon frontière dépend de ce

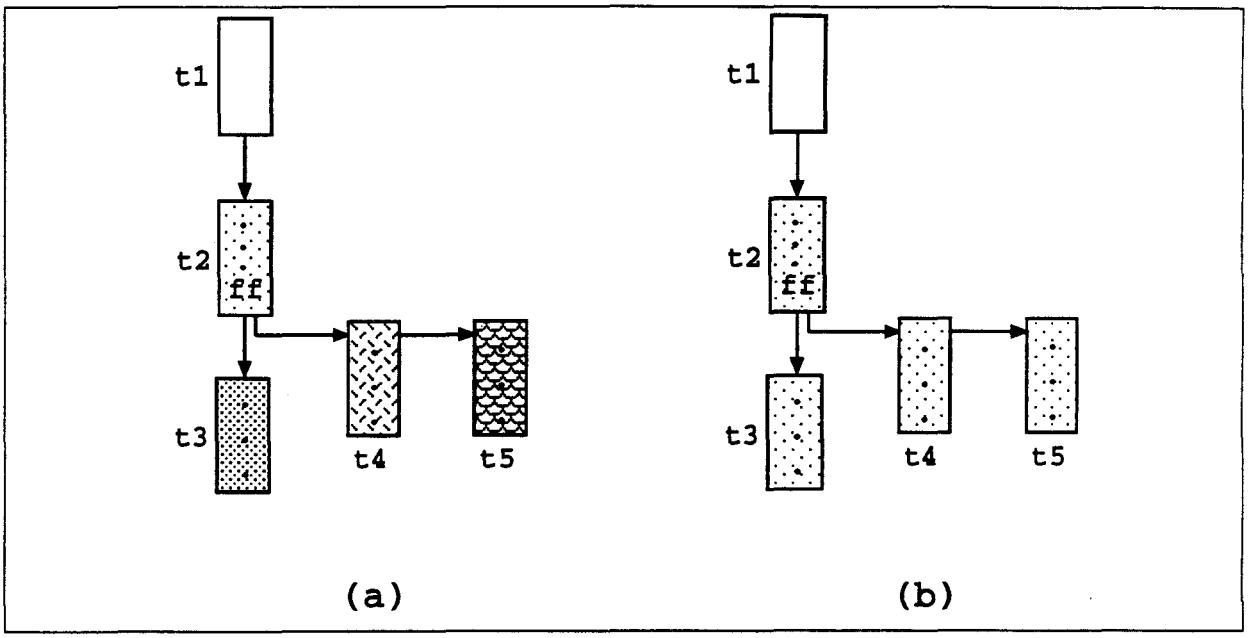



Figure 4.22 : Regroupement : Cas d'une fonction successeur d'une forme fonctionnelle génératrice de parallélisme

que cette fonction est stricte ou non stricte. Si la fonction est non stricte alors le tronçon qu'elle définit ne doit pas être un tronçon frontière. Dans le cas contraire, le problème se ramène à l'étude des conditions d'exploitation du parallélisme vertical inhérent à cette fonction non stricte. Ce problème a été traité ci-dessus. Dans la figure 4.22(b), le parallélisme inhérent à la forme fonctionnelle *ff* est supposé ne pas être exploité à l'exécution puisque les tronçons *t4* et *t5* sont remplis avec le même motif. C'est pourquoi le tronçon t3 doit être dans le même paquet que celui qui contient t2, c'est à dire que t3 n'est pas un tronçon frontière.

4. **Cas d'une fonction successeur d'une fonction d'ordre supérieur**
   La décision de considérer le tronçon identifié par une fonction frontière successeur d'une fonction d'ordre supérieur dépend de deux facteurs :

   - Le type de la fonction frontière : stricte ou non stricte ;
   - Le partitionnement en paquets appliqué à l'arborescence fonctionnelle produite à l'exécution de la fonction d'ordre supérieur. Ce facteur fait que la décision du regroupement est prise de façon dynamique.

   Dépendamment du premier facteur, le regroupement des tronçons en paquets peut être décidé statiquement à la compilation comme il peut être décidé dynamiquement à l'exécution. En effet, si la fonction frontière est stricte alors la décision de regroupement sera prise dynamiquement sur la base du deuxième facteur. Le tronçon défini par la fonction frontière sera un tronçon frontière si tous les tronçons de l'arborescence fonctionnelle produite à l'exécution de la fonction d'ordre supérieur sont regroupés dans plus d'un paquet. Dans le cas contraire, le tronçon en question ne doit pas être un tronçon frontière.

Dans le cas où la fonction frontière est non stricte, le problème se ramène à l'étude du premier cas traité ci-dessus. Si cette étude révèle que le tronçon doit être un tronçon frontière alors cette décision de regroupement est prise statiquemnent sinon celle-ci est reportée à l'exécution.

Dans le paragraphe suivant, nous allons illustrer les différentes étapes de la transformation de programme sur l'exemple de la multiplication de matrices.

## 4.7 Exemple récapitulatif : Le produit matriciel

Le programme Graal qui implémente le produit matriciel peut être écrit :

$PM \equiv \{(distr\ (distl\ PS))\ \ 1\ \ (\alpha\ list)\ o\ 2\}$
$PS \equiv add\ o\ (\alpha\ mul).$

En remplaçant dans ce programme les fonctions de distribution par leurs fonctions équivalentes en FP, on obtient le programme suivant :

$PM \equiv \{(\alpha)_{FP}\ ((\alpha)_{FP}\ PS)\ o\ (distr)_{FP}\ \ 1\ \ ((\alpha)_{FP}\ list)\ o\ 2\}$
$PS \equiv (add\ o\ ((\alpha)_{FP}\ mul))\ o\ (trans)_{FP}.$

Les étapes de transformation que subit le programme ci-dessus sont résumées ci-dessous.

### 4.7.1 Découpage du programme en tronçons

Les arborescences de tronçons issues du découpage sont illustrées par des rectangles en pointillés sur la figure 4.23. Dans cette figure, les tronçons sont identifiés de la manière suivante :

- $p3\_11$ et $p3\_61$ sont des tronçons car ils sont racines d'arborescences fonctionnelles ;
- $p3\_12, p3\_21, p3\_22, p3\_41, p3\_51$ et $p3\_62$ sont des tronçons car ils sont racines de paramètres de formes fonctionnelles ;
- $p3\_31$ et $p3\_63$ sont des tronçons car ils sont successeurs de formes fonctionnelles.

### 4.7.2 Calcul du descripteur de fenêtre associé à chaque tronçon

Les descripteurs associés aux différents tronçons identifiés à l'étape précédente sont donnés par les parties gauches des règles suivantes.

- $\tilde{p3\_11} : \epsilon \rightarrow \epsilon$
- $\tilde{p3\_12} : b^2 \rightarrow b$
- $\tilde{p3\_21} : bdbd^* \rightarrow bd^*$
- $\tilde{p3\_22} : \epsilon \rightarrow \epsilon$
- $\tilde{p3\_31} : b^2d^* + bd \rightarrow bd^*bd$
- $\tilde{p3\_41} : bdbd^* \rightarrow bd^*bd$
- $\tilde{p3\_51} : \epsilon \rightarrow \epsilon$
- $\tilde{p3\_61} : bd^*bd^* \rightarrow bd^*bd^*$

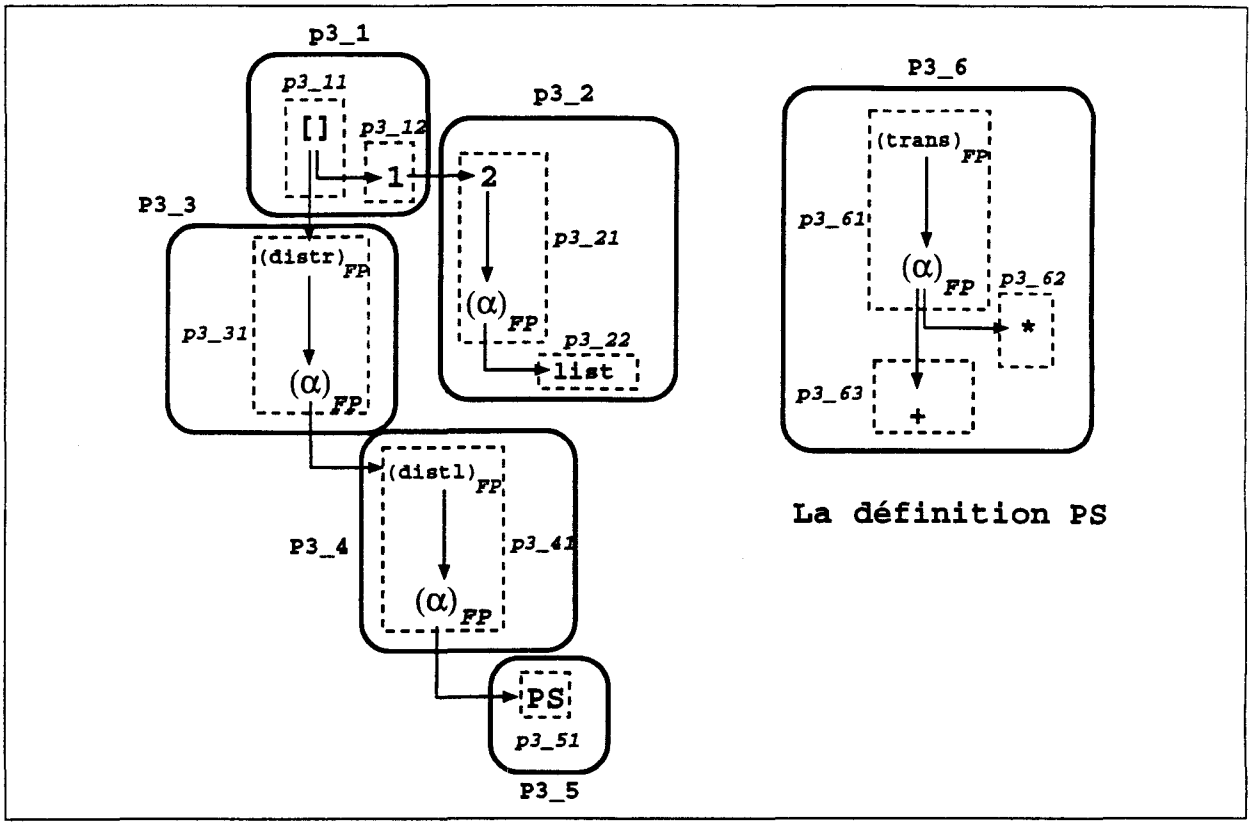



Figure 4.23 : Illustration de la transformation de programme sur le produit matriciel

- $p3\tilde{\_}62 : bd^* \rightarrow \epsilon$
- $p3\tilde{\_}63 : bd^* \rightarrow \epsilon$

### 4.7.3 Regroupement des segments en paquets

Les différents tronçons avec leurs descripteurs respectifs forment des segments. Le regroupement des différents segments en paquets est illustré dans la figure 4.23 par des boîtes en gras. Ce regroupement est obtenu par application des critères définis dans le paragraphe 4.6.

Dans le paquet $p3\_1$, le tronçon $p3\_12$ est regroupé avec le tronçon $p3\_11$ car on est dans le cas où le paramètre d'une forme fonctionnelle contient un seul paquet contenant, à son tour, un seul tronçon.
A l'inverse, $p3\_2$ est un paquet à part entière car on est dans la situation où l'arborescence fonctionnelle représentant un paramètre d'une forme fonctionnelle contient plus d'un tronçon. Dans ce paquet, le tronçon $p3\_22$ est regroupé avec le tronçon $p3\_21$ pour les mêmes raisons justifiant le regroupement de $p3\_12$ et de $p3\_11$.
$p3\_3$ constitue un paquet car le tronçon $p3\_31$ qui l'identifie est un tronçon frontière puisque la forme fonctionnelle (i.e. la construction) qui le précède est génératrice de parallélisme.
Le tronçon $p3\_41$ est également un tronçon frontière car le paramètre de la forme fonctionnelle qui le précède (ici $(\alpha)_{FP}$) contient plus d'un paquet. C'est pourquoi il identifie le paquet $p3\_4$.

$p3\_5$ est un paquet pour les mêmes raisons que pour le paquet $p3\_2$.
Le tronçon $p3\_61$ identifie un nouveau paquet i.e. $p3\_6$ car il est racine d'une définition. Le tronçon $p3\_62$ est regroupé avec $p3\_61$ car il est unique dans le paramètre de la forme $(\alpha)_{FP}$. Le tronçon $p3\_63$ est également regroupé avec $p3\_61$ car la forme $(\alpha)_{FP}$ ne génère pas, a priori, de parallélisme.

La quatrième étape de la transformation consiste à produire le code $(C + PM^2)$. Cette étape sera présentée dans le chapitre suivant.

## 4.8 Conclusion

Dans ce chapitre, nous avons proposé une nouvelle approche de gestion de la granularité dans le cadre du modèle $P^3$. Cette approche consiste en une transformation du programme Graal qui conduit à son partitionnement et naturellement au partitionnement de sa donnée associée. La transformation comporte quatre étapes.

La première étape de la transformation effectue un découpage du programme en tronçons de façon à identifier tout le parallélisme exploitable dans le programme. Les critères de découpage sont liés à la sémantique des fonctions. Le parallélisme vertical est principalement identifié par les fonctions non strictes. Par contre, le parallélisme horizontal est identifié par la présence de formes fonctionnelles. Dans la deuxième phase de la transformation, pour chaque tronçon on calcule la donnée minimale recouvrant la donnée dont elle a effectivement besoin pour s'exécuter. Cette donnée constitue la fenêtre du tronçon. Les tronçons et leurs fenêtres associées constituent des segments. La troisième étape de la transformation consiste à regrouper les segments en paquets afin d'éliminer le parallélisme jugé, a priori, inefficace. La quatrième et dernière étape de la transformation est une production de modules à partir des paquets construits à l'étape précédente. Un module est produit par traduction de la partie code d'un paquet dans un langage distribué.

Le chapitre suivant décrit la nouvelle optique d'implantation induite par la nouvelle approche de gestion de la granularité décrite ci-dessus.

# Chapitre 5

# Vers une implantation du modèle $P^3$

## 5.1 Introduction

Les trois premières étapes de transformation présentées dans le chapitre précédent (figure 4.4) permettent d'obtenir un programme sous forme d'une forêt d'arborescences de paquets. La quatrième étape de la transformation consiste à produire un code compilable multithreadé pour architectures MIMD sans mémoire commune. Ces différentes étapes conduisent à un grossissement de la granularité du programme et une nouvelle optique d'implantation du modèle $P^3$.

Pour pouvoir produire ce code, il est nécessaire d'identifier ce qui est standard (ne change pas quelque soit le programme) dans le modèle d'implantation et ce qui doit être produit (cette partie dépend du programme en entrée). Par ailleurs, la production de code parallèle d'un programme nécessite la définition de mécanismes pour l'exploitation du parallélisme, la gestion des copies et l'ordonnancement.

L'organisation de la suite de ce chapitre est la suivante : nous présentons brièvement l'environnement d'implantation du modèle $P^3$. Ensuite, nous décrvons le modèle d'implantation. Puis, nous donnons une description de la production de code ($C + PM^2$) d'un programme fonctionnel et nous l'illustrons par un exemple. Après, nous montrons comment le parallélisme horizontal et le parallélisme vertical sont exploités. Ensuite, nous décrivons le mécanisme de copie utilisé. Enfin, avant de conclure nous donnons la stratégie d'ordonnancement utilisée.

## 5.2 L'environnement d'implantation

Le modèle d'architecture, support physique de l'implantation du modèle $P^3$ est une machine MIMD sans mémoire commune. L'environnement de programmation utilisé est un environnement supportant le multithreading. Dans ce paragraphe, nous présentons l'architecture et l'environnement de programmation choisis pour le moment pour l'implantation du modèle $P^3$.

### 5.2.1 L'architecture sous-jacente

L'architecture sous-jacente à notre implantation est une ferme de 16 processeurs DEC/ALPHA 21064 (133 MHz, 64 MégaOctets de RAM et 1GigaOctet de disque) opérant sous OSF-1. Les processeurs peuvent communiquer via le réseau d'interconnexion *Gigaswitch* dédié et rapide. Ils peuvent communiquer également via le réseau classique Ethernet (10 Mbps). La figure 5.1 illustre l'architecture.

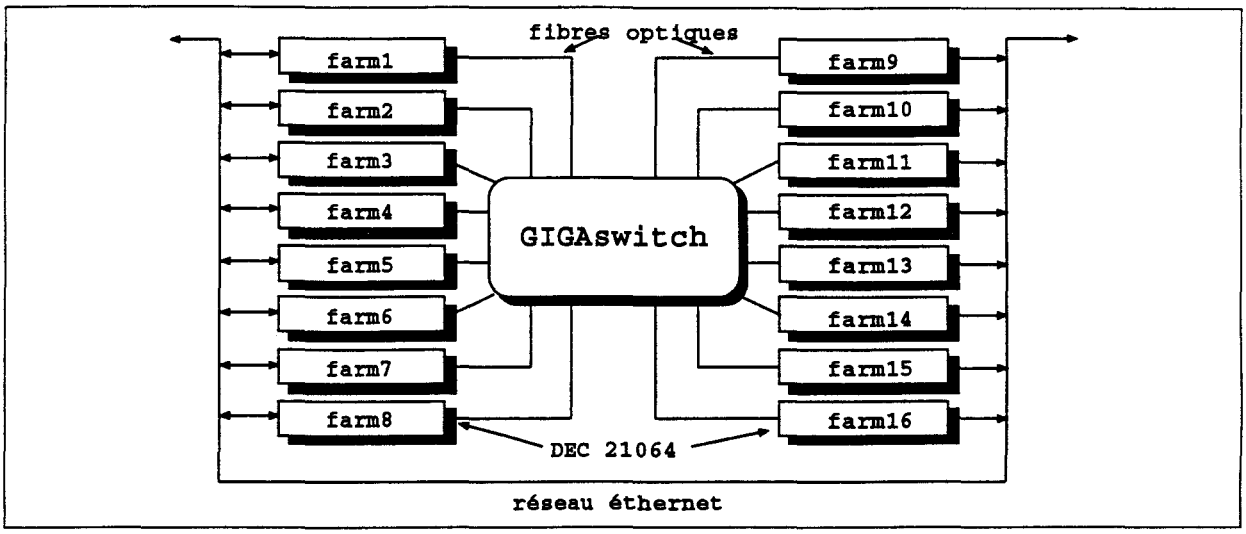



Figure 5.1 : La ferme ALPHA

### 5.2.2 L'environnement de programmation

#### 5.2.2.1 L'approche multi-tâches et l'approche multi-threads

La programmation distribuée utilise traditionnellement une approche à base de tâches. Dans cette approche, l'unité d'exécution manipulée est une tâche, appelée également *processus lourd.* Les processus UNIX et les tâches Hélios [pPS89] sont deux exemples de processus lourds.

L'utilisation de l'approche multi-tâches est de plus en plus encouragée depuis l'émergence de certains environnements tels que PVM [GBea94] et MPI [for94]. Cependant, cette approche présente un certain nombre de problèmes particulièrement lors de la programmation d'applications irrégulières. Ces problèmes sont dûs au fait que le contexte d'exécution d'un processus lourd est volumineux, et sont principalement :

- La création et le changement de contexte d'un processus lourd sont coûteux en temps de traitement ;
- La communication inter-processus utilise des canaux spéciaux (pipes, sockets, boîtes à lettres) qui sont également coûteux en temps de traitement ;

- De lourds mécanismes sont nécessaires pour implémenter le partage de l'espace d'adressage entre les processus lourds résidant sur le même site ;

Afin de pallier les problèmes ci-dessus, une autre approche est considérée et est largement utilisée aujourd'hui. Il s'agit de l'approche à base de *threads*. Il existe plusieurs définitions du concept de thread[1], appelé aussi *processus léger*. Le contexte d'un thread (pile d'exécution, compteur ordinal, etc) est très petit en comparaison de celui d'un processus lourd. Par conséquent, les threads sont une solution aux problèmes cités ci-dessus. En effet, la création et le changement de contexte d'un thread sont beaucoup plus rapides. Par exemple, le temps de création d'un thread sous PM$^2$ est d'environ 100 microsecondes [NM95, Nam97], ce qui n'est pas énorme. De plus, un thread est toujours créé à l'interieur d'un processus lourd. Il partage donc l'espace mémoire du processus avec d'autres threads. Par conséquent, la communication et la gestion du partage de la mémoire entre threads sont moins coûteuses.

Tous ces avantages ont encouragé le développement de plusieurs environnements de programmation multithreadés. Ceux-ci appartiennent à deux classes : environnements au niveau système (noyau) [ea91] et environnements au niveau utilisateur [Chr94, NM95, FS94]. Dans notre implantation, nous avons utilisé le deuxième type d'environnement, plus exactement l'environnement PM$^2$. Le paragraphe suivant présente succintement PM$^2$.

#### 5.2.2.2 L'environnement PM$^2$

PM$^2$(**P**arallel **M**ultithreaded **M**achine)[2][NM95, Nam97] est une plate-forme bâtie au dessus de PVM [GBea94] pour l'utilisation de threads préemptifs avec priorités dans un contexte distribué. Cette plate-forme est un couplage d'une bibliothèque de primitives de gestion de threads appelée MARCEL [NM95, Nam97] qui est une extension des threads Posix, et de la bibliothèque de communication PVM.

A l'instar de PVM, une application PM$^2$ est un ensemble de tâches. Par opposition à PVM (version standard), chaque tâche PM$^2$ "est threadée". Ceci peut être vu comme une virtualisation de l'architecture. En effet, une tâche peut être vue comme un site et ses threads comme les processus du site. La coopération entre les threads de l'application est assurée par l'utilisation de LRPC ("**L**ightweight **R**emote **P**rocedure **C**all). De ce fait, le concept de base de PM$^2$ est le calcul distribué. La figure 5.2, empruntée à la thèse [Nam97], illustre le modèle PM$^2$.

Le LRPC est la primitive principale de PM$^2$. Celle-ci existe en trois versions : *synchrone*, *asynchrone* et *asynchrone avec attente différée*. Le LRPC synchrone bloque le thread qui l'exécute jusqu'à ce que le résultat attendu soit retourné. Le LRPC asynchrone permet le lancement de traitements (locaux ou distants) indépendants, pouvant éventuellement retourner des résultats au thread initiateur. Un LRPC nécessitant une attente explicite d'un résultat est appelé "LRPC avec attente différée". Cette primitive est importante. Elle permet de lancer des traitements avant d'attendre leurs résultats correspondants. Un exemple illustrant son importance est la recherche d'une solution dans un arbre de recherche. Un thread explorant un arbre peut lancer, à l'aide de cette primitive, plusieurs recherches dans différents sous-arbres avant de se mettre

[1] Par exemple, dans PM$^2$ un thread est un flot de contrôle à l'interieur d'un processus UNIX. Le thread exécute une fonction dont le code se trouve dans le segment de code du processus.

[2] PM$^2$ est developpé au Laboratoire d'Informatique Fondamentale de Lille par l'équipe GOAL

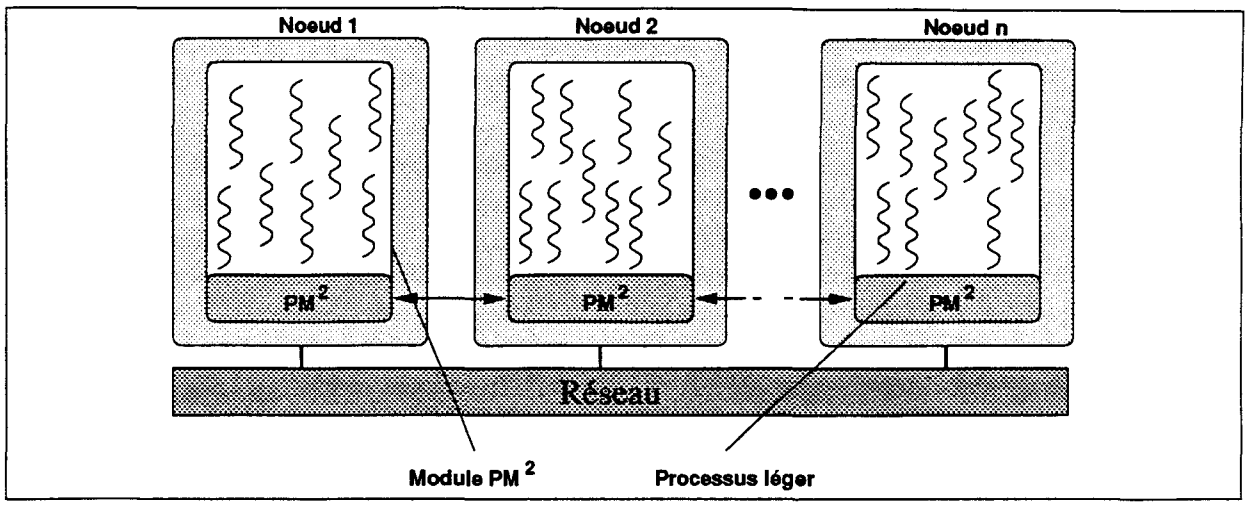



Figure 5.2 : Le modèle PM²

en attente des résultats de ces recherches. Dans [MDLT96g, MDLT96a, MDLT96d, MDLT96c], seuls les threads ayant échoué dans leurs recherches renvoient le résultat au thread père. Un thread ayant réussi dans sa recherche remonte directement la solution au maître.

PM² intègre plusieurs fonctionnalités intéressantes telles que la création à distance et la migration de threads, la gestion des priorités et l'augmentation dynamique de la taille de la pile. Ces fonctionnalités font de PM² un bon candidat pour les implantations visant l'utilisation de threads distribués. La migration et la gestion des priorités sont particulièrement séduisantes pour l'équilibrage de charge. Elles constituent les fonds d'investissement du projet LBMP (**L**oad **B**alancing with **M**igrations directed by **P**riorities) [Den96].

## 5.3 Le modèle d'implantation

Le modèle d'implantation de P³ est composé de deux tâches principales appelées *P3_START* et *P3_WORKER*.

Il existe une seule tâche P3_START. Celle-ci est créée sur le site qui lance l'exécution de l'application P³ [3]. Son rôle principal est d'initialiser l'environnement de programmation, puis de lancer les tâches P3_WORKER sur tous les sites de la machine, ensuite d'installer le régulateur de charge puis de lancer l'évaluation du programme sur une tâche P3_WORKER et enfin de terminer l'application.

Un exemplaire de la tâche P3_WORKER est créé sur chaque site. Son rôle essentiel est de coopérer, par un ensemble de services, avec les autres tâches P3_WORKER pour assurer l'exécution de l'application. Elle effectue également les opérations de régulation de charge.

Les définitions des deux tâches P3_START et P3_WORKER utilisent deux fichiers appelés "p3_rpc.h" et "p3_rpc.c". Le fichier "p3_rpc.h" contient principalement la liste des services util-

[3] Une application P³ est une application exécutée selon le modèle P³

isés par les deux tâches ainsi que les déclarations des paramètres d'entrée et des résultats de chaque service utilisé. Le fichier "p3_rpc.c" correspondant au fichier "p3_rpc.h" contient les primitives d'empaquetage et de dépaquetage des paramètres d'entrée et des résultats de chaque service. Les versions $PM^2$ des deux fichiers sont données ci-dessous.

**a) Le fichier "p3_rpc.h"**

```
#include <pm2.h>

/* Liste des services du modèle d'implantation */
BEGIN_LRPC_LIST
  LRPC_P3_module
      ...
END_LRPC_LIST

/* Nom du module à exécuter par le service LRPC_P3_module */
typedef char p3ModuleName[8];

LRPC_DECL_REQ(LRPC_P3_module, p3ModuleName p3Module;
              struct DATANODE *targetNode;)
LRPC_DECL_RES(LRPC_P3_module,)
```

Dans le fichier ci-dessus, nous n'avons pas mis tous les services utilisés par le modèle d'implantation pour éviter d'encombrer ce dernier. En effet, il existe d'autres services tels que le service "LRPC_INSTALL" qui permet la réception, en provenance de la tâche P3_START, de la table "workers" qui contient les identités des threads gestionnaires de toutes les tâches P3_WORKER. Les services utilisés pour la régulation de charge seront donnés dans la partie III de cette thèse.

Le service "LRPC_P3_module" est le service principal du programme. Il exécute le code correspondant aux modules de l'application. Les paramètres d'entrée de ce service représentent respectivement le nom de la fonction (d'un paquet ou module) à exécuter par le service et le nœud cible correspondant. La définition de ce service sera donnée dans la tâche P3_WORKER.

**b) Le fichier "p3_rpc.c"**

```
#include "p3_rpc.h"

PACK_REQ_STUB(LRPC_P3_module)
  pvm_pkstr(arg->p3Module);
  pvm_pkpointer(&arg->targetNode,1,1);
END_STUB

UNPACK_REQ_STUB(LRPC_P3_module)
  pvm_upkstr(arg->p3Module);
  pvm_upkpointer(&arg->targetNode,1,1);
END_STUB
```

```
PACK_RES_STUB(LRPC_P3_module)
END_STUB

UNPACK_RES_STUB(LRPC_P3_module)
END_STUB
```

### 5.3.1 La tâche P3_START

Le code de la tâche P3_START est contenu dans un fichier appelé "p3_start.c". La version sous PM² de ce fichier est la suivante :

```
#include "p3_rpc.h"
#include "p3_mods.h"

LRPC_REQ(LRPC_INSTALL) workers;
LRPC_REQ(LRPC_P3_module) req;
struct DATANODE *data;

main()
{
  short worker0=workers[0];

  /* Initialisation du système des rpc */
  pm2_init_rpc();

  /* Initialisation de pm2 */
  pm2_init();

  /* Création des tâches P3_WORKER */
  create_workers();

     ...

  /* Saisie de la donnée du programme */
  data=input_data();

  /* Préparation des arguments de LRPC_P3_module */
  req.p3Module=``p3_1";
  req.targetNode=data;

  /* Lancement du programme (module racine) par un LRPC asynchrone */
  ASYNC_LRPC(worker0, LRPC_P3_module, p2, DEFAULT_STACK, &req);

  pm2_exit();
}
```

Le premier module du programme est lancé par un LRPC asynchrone de type "LRPC_P3_module". Les paramètres du service sont respectivement le "site" qui exécutera le service, le type du service, la priorité du thread qui va exécuter le service, la taille de la pile de celui-ci et les arguments du service. Pour le lancement du premier module (thread), le site est choisi au hasard. La priorité est fixée à la valeur "p2". Celle-ci sera donnée dans le paragraphe 5.7 sur l'ordonnancement.

### 5.3.2 La tâche P3_WORKER

Le code de la tâche P3_WORKER est contenu dans le fichier "p3_work.c". La version sous $PM^2$ de ce fichier est donnée ci-dessous.

```
#include "p3_rpc.h"
#include "p3_mods.h"

LRPC_SERVICE(LRPC_P3_module)
  arg.p3Module(arg.targetNode);
END_SERVICE(P3_module)

main()
{
  /* Initialisation du système des rpc */
  pm2_init_rpc();

  /* Déclaration des services à exécuter par la tâche */
  DECLARE_LRPC(LRPC_P3_module);

     ...

  /* Initialisation de pm2 */
  pm2_init();

  pm2_exit();
}
```

Comme on pourra le voir, dans le paragraphe suivant, à travers l'exemple du produit matriciel, la traduction d'un programme dans ($C + PM^2$) produit une librairie de fonctions appelée "p3_mods" qui implémente les différents modules du programme. Cette librairie utilise une autre librairie appelée "p3_graal", où sont définies toutes les fonctions du langage Graal. La librairie "p3_mods" est inclue dans la tâche P3_WORKER. Le code du programme est donc dupliqué sur chaque site, ce qui fait que l'association du code avec sa donnée pendant l'exécution ne pose aucun problème de localisation de celui-ci.

Le service "LRPC_P3_module" consiste à appliquer son premier paramètre à son deuxième paramètre c'est à dire appliquer la fonction qu'il est censé exécuter au nœud cible de celle-ci.

Le paragraphe suivant décrit la production de code d'un programme. Une illustration sur un exemple y sera également présentée.

## 5.4 Production de code (C+PM$^2$)

### 5.4.1 Description

Dans le paragraphe 4.3.1, nous avons identifié quatre étapes de transformation d'un programme avant son exécution. Les trois premières étapes permettent de produire une forêt d'arborescences de paquets. La quatrième étape consiste à traduire la partie code de chacun des différents paquets du programme dans $(C + PM^2)$ et produire le code associé à tout le programme.

La production de code se fait de façon récursive en commençant par le paquet racine de l'arborescence principale. Le parcours des arborescences se fait en largeur d'abord.

La production du code associé à chaque paquet consiste en trois phases.

- La première phase produit le code correspondant à chacun des paquets avec lesquels le paquet courant a une liaison par l'un des trois arcs : "$\downarrow$", "$\rightarrow$" et "$\hookrightarrow$". Si un paquet contient un appel à une définition, le code associé au paquet racine de celle-ci doit être produit.

- La deuxième phase produit le code correspondant à chacun des tronçons du paquet. Ces derniers forment une arborescence. La production du code des différents tronçons est également récursive et commence à la racine de l'arborescence. Le parcours de l'arborescence se fait en largeur d'abord.
  Pour chaque tronçon, on produit une fonction. Deux cas particuliers doivent être considérés. D'une part, si un tronçon est réduit à une seule fonction alors il n'est pas nécessaire de produire son code, on utilisera directement la fonction elle-même. D'autre part, si un tronçon est unique dans un paquet alors il n'est pas nécessaire de lui définir une fonction. Son code est utilisé directement dans le paquet (c'est le code du paquet).

- La troisième phase produit le code du paquet en utilisant le code fourni par les deux phases précédentes.

Chaque fonction implémentant un paquet est exécutée par un simple appel (à cette fonction) si celle-ci est exécutée par le thread qui déclenche son exécution. A l'inverse, si le paquet doit être exécuté en parallèle avec le thread qui déclenche son exécution alors la fonction associée au paquet est exécutée par un autre thread. Par conséquent, il est nécessaire de définir chaque fonction sous forme de service. Le service a pour rôle d'exécuter la fonction soit localement ou à distance. Pour ce faire, nous avons défini le service "LRPC_P3_module" pour exécuter n'importe quelle fonction. Il suffit de lui passer en paramètre le nom de la fonction à exécuter.

Lors de l'écriture des fonctions des librairies "p3_graal" et "p3_mods", un problème important doit être considéré. Il s'agit de la manière avec laquelle la notion de fenêtre cible, utilisée à la troisième étape de la transformation, doit être exploitée.

Les motivations principales de la définition de la notion de fenêtre sont :

- Regrouper la donnée réellement utilisée par une fonction (simple, un tronçon ou un paquet) de façon à augmenter le degré de localité des données ;

- Eclater la donnée réellement utilisée par une fonction (simple, un tronçon ou un paquet) de façon à augmenter le degré de parallélisme inhérent aux données.

Ce regroupement/éclatement de la donnée **guidé par la notion de fenêtre** est fait *lors des copies des données.* Le mécanisme de copie et ses avantages seront présentés dans le paragraphe 5.6. Par ailleurs, une fois que les données sont copiées, l'association entre le code du programme (se trouvant sur tous les sites) et ses données est faite par des appels procéduraux.

### 5.4.2 Exemple : Le produit matriciel

Dans le paragraphe 4.7 du chapitre précédent, nous avons illustré la transformation de programme sur l'exemple du produit matriciel. La librairie "p3_mods" produite à la traduction de ce programme dans ($C + PM^2$) est stockée dans le fichier "p3_mods.c" donné ci-après.

```
#include "p3_rpc.h"
#include "p3_graal.h"
#include "p3_mods.h"

void p3_61(targetNode);
  struct DATANODE *targetNode;
{
  p3_transFP(targetNode);
  p3_alphaFP(p3_mul,targetNode);
}

void p3_6(targetNode);
  struct DATANODE *targetNode;
{
  p3_61(targetNode);
  p3_add(targetNode);
}

void p3_5(targetNode);
  struct DATANODE *targetNode;
{
  p3_6(targetNode);
}

void p3_4(targetNode);
  struct DATANODE *targetNode;
{
  p3_distlFP(targetNode);
  p3_alphaFP(p3_5,targetNode);
}
```

```
void p3_3(targetNode);
  struct DATANODE *targetNode;
{
  p3_distrFP(targetNode);
  p3_alphaFP(p3_4,targetNode);
}

void p3_2(targetNode);
  struct DATANODE *targetNode;
{
  p3_select(2,targetNode);
  p3_alphaFP(p3_list,targetNode);
}

void p3_1(targetNode);
  struct DATANODE *targetNode;
{
  LRPC_REQ(LRPC_P3_module) req;
  pm2_rpc_wait att;

  p3_constructFP(targetNode);
  /* Préparation des arguments de LRPC_P3_module */
  req.p3Module="p3_2";
  req.targetNode=right_brother(son(targetNode));
  LRP_CALL(select_node(), LRPC_P3_module, p2, DEFAULT_STACK,
           &req, NULL, &att);
  p3_select(1,targetNode);
  LRP_WAIT(&att);
}
```

Dans la fonction "p3_1", le deuxième paramètre de la fonction $([])_F P$ est exécuté par un LRPC synchrone avec attente différée. Les paramètres de cet appel sont respectivement l'identité du "site" qui exécutera le LRPC, le nom du service, la priorité du thread qui va exécuter le service, la taille de la pile du thread, les arguments du service, ses résultats et le sémaphore sur lequel le thread initiateur doit attendre le "call back". L'identité du site est déterminée par la fonction "select_node" qui fait appel au régulateur de charge. Par ailleurs, la structure "req" contient les arguments du service, à savoir la fonction exécutée par le service appelé i.e. la fonction "p3_2" et le nœud cible de celle-ci.

## 5.5 Exploitation du parallélisme

### 5.5.1 Parallélisme horizontal

Le parallélisme horizontal concerne principalement les formes fonctionnelles génératrices de parallélisme et les fonctions polyadiques. Dans le paragraphe 4.6.2, nous avons vu que ces sources

de parallélisme peuvent être exprimées à l'aide des fonctions $\alpha$ et [] du langage FP. Nous allons étudier l'exploitation du parallélisme en présence de ces deux formes et en présence de fonctions polyadiques.

### a) Cas de $(\alpha)_{FP}$

Nous rappelons que la forme fonctionnelle $(\alpha)_{FP}$ est définie par :

$$(\alpha)_{FP}\ \mathrm{P} : < d_1, d_2, ..., d_k > \equiv < P : d_1, P : d_2, ..., P : d_k >$$

où $< d_1, d_2, ..., d_k >$ est une séquence d'arguments et P est une fonction donnée.

Nous avons vu que l'exploitation du parallélisme inhérent à la forme $(\alpha)_{FP}$ dépend du découpage appliqué à son paramètre. En effet, si ce dernier comporte plusieurs paquets ou un paquet contenant plusieurs tronçons alors les applications $(P : d_i)_{(i=1..k)}$ se font en parallèle. Dans le cas contraire, leur exécution est séquentielle. De ce fait, deux implémentations sont prévues pour la forme $(\alpha)_{FP}$ dans le modèle d'implantation : une *version séquentielle* et une *version parallèle.* Par exemple, dans la figure 4.23, la version séquentielle doit être utilisée pour la forme $(\alpha)_{FP}$ apparaissant dans les paquets $p3\_2$ et $p3\_6$. Par contre, la version parallèle doit être utilisée pour la forme $(\alpha)_{FP}$ appartenant aux paquets $p3\_3$ et $p3\_4$.

#### Version séquentielle

Dans la version séquentielle de la forme $(\alpha)_{FP}$ (ex. $(\alpha)_{FP}list$), si toute la séquence d'arguments se trouve sur le même site (la longueur de la liste est inférieure à un certain seuil) que la forme alors l'exécution de cette dernière est séquentielle et locale. A l'inverse, si la séquence est répartie sur plus d'un site alors l'exécution est séquentielle et distribuée. Il faut noter que, dans ce cas précis, le parallélisme inhérent à la fonction $(\alpha)_{FP}$ n'est pas important, c'est pourquoi son exploitation n'est pas intéressante. En effet, par exemple, l'exploitation du parallélisme de $(\alpha)_{FP}list$ se fait de la façon suivante : avant d'exécuter chaque sous-séquence $ss1$ de l'argument, on lance l'exécution de la sous-séquence suivante $ss2$ se trouvant sur un autre site. Il est alors nécessaire de parcourir $ss1$ jusqu'à son dernier nœud pour pouvoir trouver l'adresse de la $ss2$. De plus, un autre parcours est nécessaire pour exécuter $ss1$. Par conséquent, il faut deux parcours de toute la séquence pour pouvoir exploiter le parallélisme alors que l'exécution de la fonction *list* est une simple mise à jour d'une information. Elle peut donc être faite en utilisant un seul parcours. Dans le premier cas, l'exécution se fait par un simple parcours de la séquence. Par contre, dans le deuxième cas l'exécution fait intervenir des LRPC sous PM$^2$. La figure 5.3 illustre un exemple d'une telle situation.

Dans cet exemple, deux LRPC (LRPC1 et LRPC2) de type "LRPC_P3_module" sont nécessaires pour effectuer les exécutions distribuées du paramètre sur les deux fragments se trouvant sur les sites 2 et 3. Le traitement de chacun des deux LRPC consiste en une application séquentielle du paramètre de la forme fonctionnelle aux arguments de la sous-séquence correspondante. LRPC1 et LRPC2 sont asynchrones. Pour assurer la synchronisation, le thread qui exécute la forme $(\alpha)_{FP}$ doit se bloquer sur un sémaphore juste après le lancement de LRPC1. Il sera débloqué par le thread qui traite le dernier fragment de la séquence en effectuant l'appel LRPC3 sur le site qui contient la forme $(\alpha)_{FP}$.

Afin de pouvoir effectuer cet appel, il est nécessaire d'utiliser comme paramètre dans LRPC1

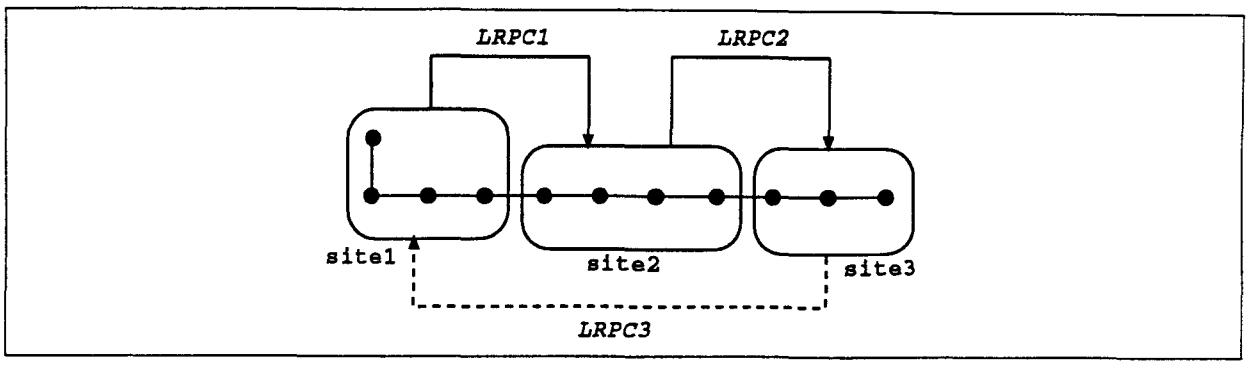



Figure 5.3 : Version séquentielle et distribuée de la forme $(\alpha)_{FP}$

et LRPC2 le numéro du site qui exécute la forme $(\alpha)_{FP}$. Ainsi, "LRPC_P3_module" doit avoir trois paramètres : le nom de la fonction à exécuter, le nœud cible de la fonction et le numéro du site qui exécute la forme $(\alpha)_{FP}$. Dans le cas où ce dernier paramètre est inutile, il est mis à la valeur "NULL".

### Version parallèle

Dans la version parallèle de la forme $(\alpha)_{FP}$, le parallélisme exploité est de type "pipe-line". En effet, si la séquence d'arguments est répartie en fragments (sous-séquences) sur plusieurs sites alors le traitement de chaque fragment commence d'abord par le lancement du traitement du fragment frère droit. Par exemple, dans la figure 5.4, le traitement du premier fragment commence par l'appel LRPC1 de type "LRPC_P3_module" qui permet de lancer le traitement du fragment suivant. De même, ce dernier commence par l'appel LRPC2 qui lance le traitement du dernier fragment. Les appels LRPC1 et LRPC2 sont des LRPC asynchrones avec attente différée. De ce fait, la synchronisation est assurée en cascade : le LRPC1 ne peut renvoyer son rappel ("call back") qu'après avoir reçu le rappel de LRPC2.

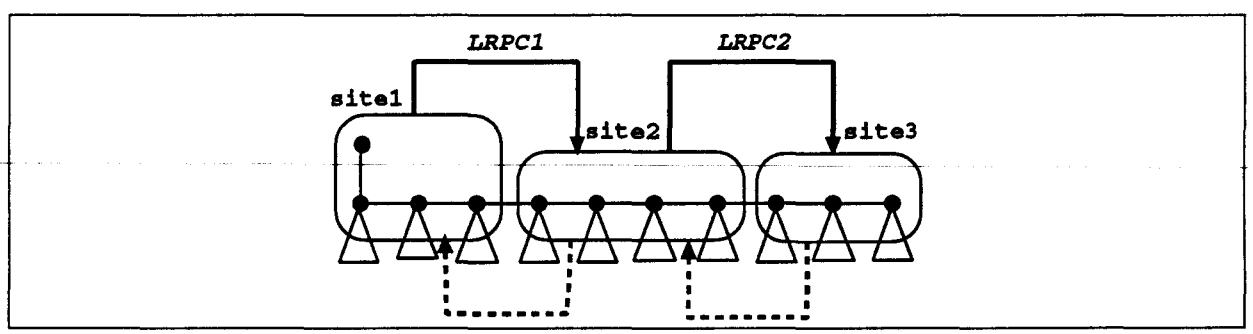



Figure 5.4 : Version parallèle de la forme $(\alpha)_{FP}$

Le traitement de chaque segment consiste dans un premier temps, comme il a été dit ci-dessus, à lancer le traitement du fragment suivant dans la séquence. Ensuite, on applique le paramètre

de la forme fonctionnelle sur les nœuds successifs de la sous-séquence. Le parcours de celle-ci est séquentiel et l'exécution du paramètre de la forme $(\alpha)_{FP}$ sur chaque nœud parcouru se fait suivant la répartition de la donnée dont ce dernier est racine.

### b) Cas de $([])_{FP}$

Nous rappelons que la forme fonctionnelle $([])_{FP}$ est définie par :

$$([P_1, P_2, ..., P_k])_{FP} : \mathrm{d} \equiv < P_1 : d, P_2 : d, ..., P_k : d >$$

où $P_1$, $P_2$, ..., $P_k$ sont des fonctions et d une donnée.

L'exécution de la forme fonctionnelle $([])_{FP}$ nécessite autant de copies de l'argument qu'elle a de paramètres. Le mécanisme de copie sera décrit dans le paragraphe 5.6.

Par ailleurs, nous avons vu dans le paragraphe 4.6.2 que l'exploitation du parallélisme inhérent à la forme $([])_{FP}$ dépend du découpage appliqué à ses paramètres. Si un paramètre est regroupé avec la forme fonctionnelle alors il est exécuté localement. Par exemple, dans la figure 4.23, le premier paramètre de la forme $([])_{FP}$ du paquet $p3\_1$ est exécuté localement i.e. sur le site qui contient la forme fonctionnelle.
Si au contraire, le paramètre ne doit pas être, a priori, regroupé avec $([])_{FP}$ alors il sera exécuté suivant la répartition de la copie qui lui a été créée.

### c) Cas des fonctions polyadiques

Les arguments d'une fonction polyadique peuvent être des données simples ou des fonctions. Dans ce dernier cas, nous avons vu dans le paragraphe 7.2 que la fonction polyadique peut être exprimée à l'aide des formes fonctionnelles $(\alpha)_{FP}$ et $([])_{FP}$. L'étude de l'exploitation du parallélisme des fonctions polyadiques dans ce cas se ramène à l'étude de l'exploitation du parallélisme en présence des deux formes fonctionnelles.
Dans le cas où les arguments sont des données, la fonction polyadique est exécutée séquentiellement.

## 5.5.2 Parallélisme vertical

Nous rappelons que le parallélisme vertical, dit également d'anticipation, est exploité en présence de fonctions non strictes. A l'exécution, ce type de parallélisme se traduit de la façon suivante : un thread exécutant un module lance, en même temps, un autre thread pour l'exécution du module successeur. Un module est dit successeur d'un autre module s'ils sont liés dans le programme par le symbole "↓".

L'exploitation de ce type de parallélisme en utilisant plusieurs sites (parallélisme effectif) induit un surcoût de communication car deux threads exécutant deux modules successifs (liés par le symbole "↓") ont le même nœud cible (leur donnée est sur le même site). Par conséquent, l'exploitation du parallélisme vertical est locale dans notre modèle. Il s'agit d'un pseudo-parallélisme (Multithreading sur un site). Le type d'appel de service utilisé est le LPRC asynchrone.

Le partage d'un même nœud cible par deux threads différents pose un problème de synchronisation. La gestion de ce problème est assurée par utilisation de sémaphores. Le déblocage d'un thread bloqué sur un nœud cible est à la charge de l'ordonnanceur MARCEL de PM².

## 5.6 Gestion des copies

Afin d'exploiter le parallélisme inhérent à certaines fonctions du langage Graal, telles que les fonctions de distribution, la forme fonctionnelle "while", la forme fonctionnelle construction, ...etc, l'argument doit être copié. Dans ce paragraphe, nous allons étudier le mécanisme de copie à travers deux exemples différents qui nous semblent révélateurs de certains problèmes.

### Cas des fonctions de distribution : La fonction $(distl)_{FP}$

L'exécution de la fonction de distribution à gauche $(distl)_{FP}$ est illustrée par la figure 5.5. La distribution à gauche de l'argument x (sous-arbre de données) sur la séquence d'arguments $y1$ $y2$ $y3$ (sous-arbres de données également) nécessite, comme le montre la figure 5.5, deux copies x' et x" de x, une pour le deuxième argument et une autre pour le troisième. L'argument original i.e. x est utilisé pour le premier argument i.e. y1.

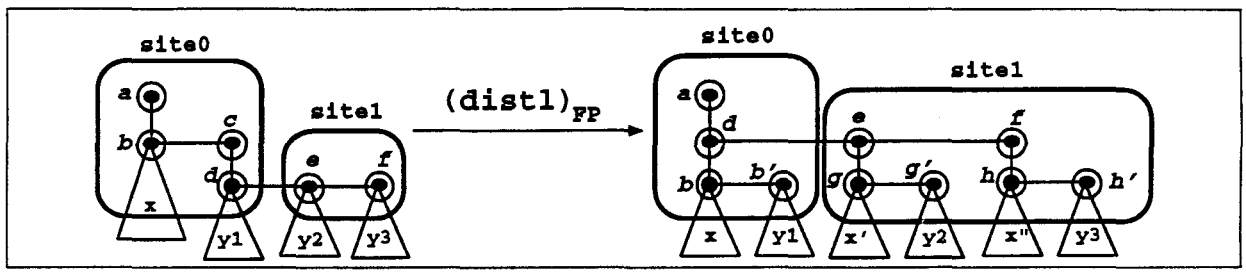



Figure 5.5 : Mécanisme de copie : Cas de la fonction $(distl)_{FP}$

Le mécanisme de copie nécessite au minimum deux types de services : un service pour l'initiation du mécanisme de copie et un service pour la demande de copie. Ces services sont appelés respectivement "LRPC_P3_Copy1" et "LRPC_P3_Copy2".

Le service "LRPC_P3_Copy1" permet d'initier le mécanisme de copie sur chacun des fragments de l'argument de la fonction $(distl)_{FP}$. Dans l'exemple de la figure 5.5, un LRPC de type "LRPC_P3_Copy1" est lancé sur le deuxième fragment i.e. celui qui contient les nœuds $e$ et $f$ de la séquence afin d'initier les copies $x'$ et $x"$. Le traitement de ce service consiste à créer les nœuds $g$ et $g'$ puis lancer la copie de $x'$, ensuite de créer les nœuds $h$ et $h'$ puis lancer la copie de $x"$. Il faut noter que la création de ces nœuds, comme le montre la figure 5.5, est locale. Ceci permet d'augmenter le degré de localité des données.
Afin de pouvoir retrouver l'argument $x$ à copier, le nom du nœud racine de celui-ci i.e. $b$ doit être un paramètre du service "LRPC_P3_Copy1". Les autres paramètres du service sont le nom du service, le nom de la fonction (i.e. $(distl)_{FP}$) et le nœud cible (i.e. $e$). Par ailleurs,

"LRPC_P3_Copy1" est de type asynchrone avec attente différée. La synchronisation est assurée par utilisation de sémaphores.

Le lancement de la copie de x se traduit par un LRPC asynchrone avec attente différée de type "LRPC_P3_Copy2". Les paramètres de ce service sont le nom du service, le nom du nœud racine de la donnée à copier (i.e. $b$ dans l'exemple de la figure 5.5) et le nom du nœud racine de la nouvelle copie (i.e. $g$ pour $x'$ et $h$ pour $x''$).

Le traitement du service "LRPC_P3_Copy2" consiste à effectuer la copie proprement dite de l'argument à copier. Dans la figure 5.5, deux appels de ce type sont faits sur le sous-arbre de données $x$. Le premier problème qui se pose est la synchronisation des deux services s'occupant de la copie $x'$ et de la copie $x''$ car ceux-ci partagent la même donnée. Cette synchronisation est assurée par utilisation de sémaphores d'exclusion mutuelle.

Le deuxième problème posé par le traitement de "LRPC_P3_Copy2" est la réalisation de la copie proprement dite de la donnée. En effet, cette dernière laisse apparaître deux grandes questions :

- Vers quel(s) site(s) envoyer la donnée à copier?
- Comment mettre en œuvre le processus de copie?

Le ou les site(s) désignés pour recevoir la copie sont sélectionnés à partir d'une table locale de sites en sous-charge. Cette table est déterminée par le régulateur de charge qui sera présenté dans la troisième partie de la thèse.

La réponse à la deuxième question dépend de deux paramètres importants : la taille de la donnée à copier et la répartition courante de celle-ci. En effet, si la donnée est de taille petite et qu'elle est entièrement groupée sur un seul site alors elle est copiée telle quelle i.e. sans modification de sa répartition. Par contre, si elle est répartie sur plus d'un site alors elle doit être copiée avec regroupement sur un seul site.
A l'inverse, si la donnée est de taille importante et qu'elle est entièrement groupée sur un seul site alors elle est copiée avec éclatement sur plusieurs sites. Par contre, si elle est répartie sur plus d'un site alors elle est copiée avec redéfinition de sa répartition.

L'éclatement/regroupement de la donnée pose les trois problèmes suivants :

- La définition d'un critère d'éclatement/regroupement ;
- La communication de la donnée copiée ;
- Le rétablissement, sur le site déstinataire de la donnée copiée, des liens entre les différents fragments de la copie.

Dans [Ben94, BM95], la technique de copie simulée utilise un seuil d'éclatement/regroupement. Les fragments résultant de l'éclatement sont obtenus par comptage de nœuds en commençant par la racine de la donnée à copier. Le parcours de la donnée (sous-arbre) se fait en profondeur d'abord et de gauche à droite. Cette technique constitue d'une part un moyen de grossissement de la granularité des données (bon degré de localité). D'autre part, il représente un

outil d'exploitation du parallélisme inhérent aux données. Ceci encourage sa réutilisation dans l'implantation.

Les sous-arbres de données constituant une donnée à copier sont aplatis avant d'être communiqués. L'aplatissage d'un sous-arbre consiste à mettre celui-ci, en le parcourant en profondeur d'abord et de gauche à droite, dans un vecteur de nœuds (enregistrements). Afin de pouvoir retrouver la structure arborescente à son arrivée sur le site destinataire, il est nécessaire de joindre au vecteur un descripteur de l'organisation des nœuds du sous-arbre. Ce descripteur est un vecteur qui contient pour chaque nœud du sous-arbre, dans l'ordre où il apparaît dans le vecteur des nœuds, deux caractères. Le premier caractère indique si le nœud a un fils et le deuxième caractère indique si le nœud a un frère droit. Trois valeurs sont possibles pour ces caractères : "0" si le nœud n'a pas de fils (ou frère), "1" si le nœud a un fils (ou frère) qui est local et "2" si le nœud a un fils (ou frère) qui est distant. La figure 5.6 illustre cette technique d'aplatissage. Les deux vecteurs représentent le vecteur de nœuds résultat de l'aplatissage de l'arbre et le descripteur associé.

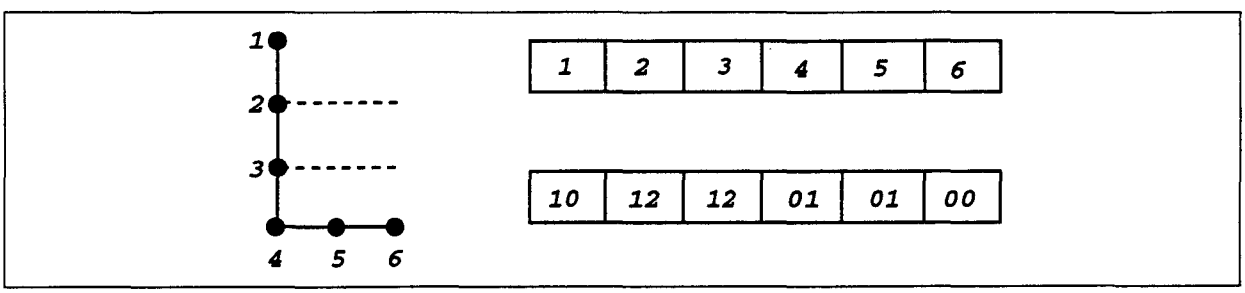



Figure 5.6 : Illustration du mécanisme d'aplatissage d'un sous-arbre de données

Le problème du rétablissement, sur le site déstinataire de la donnée copiée, des liens entre les différents fragments de la copie est analogue au problème de reconstitution d'un message à partir de ses différents paquets dans le protocole TCP/IP. Dans notre implantation, la valeur "2" du deuxième caractère associé à chaque nœud dans son descripteur est utilisée dans le but de rétablir les liens entre les fragments de la copie.

Lors de la réalisation de la copie proprement dite, d'autres services sont utilisés.

### Cas de la fonction $([])_{FP}$

Le mécanisme de copie est le même que pour la fonction précédente. L'avantage de cet exemple est qu'il met bien en évidence *une nouvelle technique de copie optimisée.* Il s'agit d'une **copie avec seuil de regroupement dirigée par la fenêtre.**

La copie dirigée par la fenêtre consiste à ne copier que la fenêtre d'un paramètre et tout ce qui est attaché aux feuilles de la fenêtre. Par exemple, dans le cas du produit matriciel (figure 4.23) le paquet *p3_1* contient la fonction $([])_{FP}$. Seule la fenêtre (i.e. *bdbd**) associée au paquet *p3_2* et les données attachées aux feuilles de cette fenêtre sont copiées i.e. la deuxième matrice. La copie de la première matrice est donc évitée, ce qui optimise la copie.

Un autre exemple illustrant l'optimisation du mécanisme de copie est celui donné dans la figure 5.7. Cet exemple montre la partie de la donnée copiée lors de l'exécution de la fonction $([P, cdr\ o\ car, car\ o\ cdr])_{FP}$ sur la donnée $x =< 0\ 1\ 2\ 3 >< 4\ 5\ 6\ 7 >< 8\ 9\ 10\ 11 >$. P est un paramètre quelconque et le sous-arbre x sur la figure 5.7 désigne la donnée x i.e. l'original de la donnée à copier. Seules les fenêtres $b^2d$ (de $(cdr\ o\ car)$) et $bd$ (de $(car\ o\ cdr)$) et les données qui leurs sont attachées sont copiées. Ce qui permet un gain considérable en espace mémoire et en temps de copie.

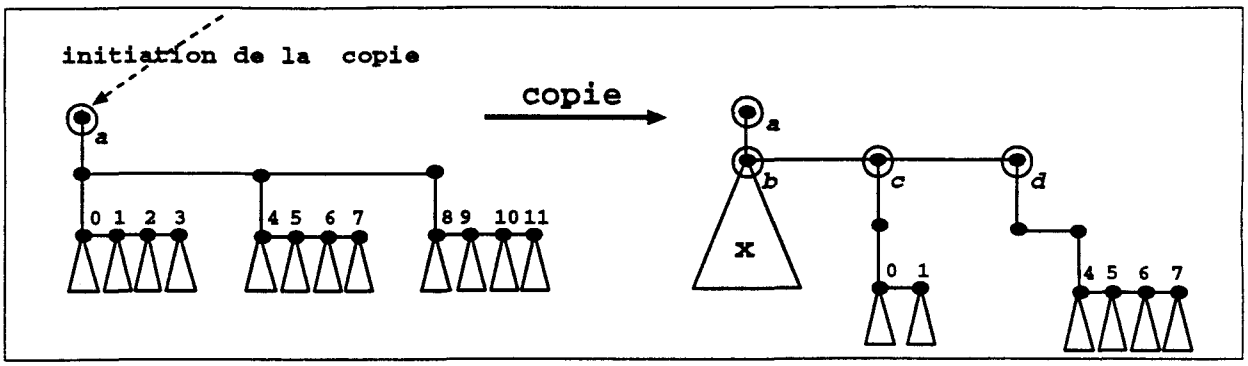



Figure 5.7 : Mécanisme de copie : Cas de la fonction $([])_{FP}$

La copie guidée par la notion de fenêtre peut être également utilisée dans le cas précédent i.e. pour la fonction $(distl)_{FP}$ sauf que ce cas nécessite une analyse supplémentaire de la fenêtre associée au paramètre de la fonction. Ceci parce que les copies créées sont attachées à d'autres sous-arbres. Il est donc nécessaire d'identifier dans la fenêtre du paramètre la partie qui appartient à la donnée copiée.

Les services utilisés dans la mise en œuvre du mécanisme de copie avec seuil dirigée par la fenêtre doivent avoir comme paramètre d'entrée le descripteur de la fenêtre du paquet racine du paramètre d'une forme fonctionnelle qui utilisera la copie. Ce descripteur (qui est une chaîne de caractères) diminue au fur et à mesure que la copie s'effectue. En effet, chaque fois qu'un fils (respectivement frère) d'un nœud est copié on supprime un "b"(respectivement "d") du descripteur. La figure 5.8 illustre la propagation du descripteur.

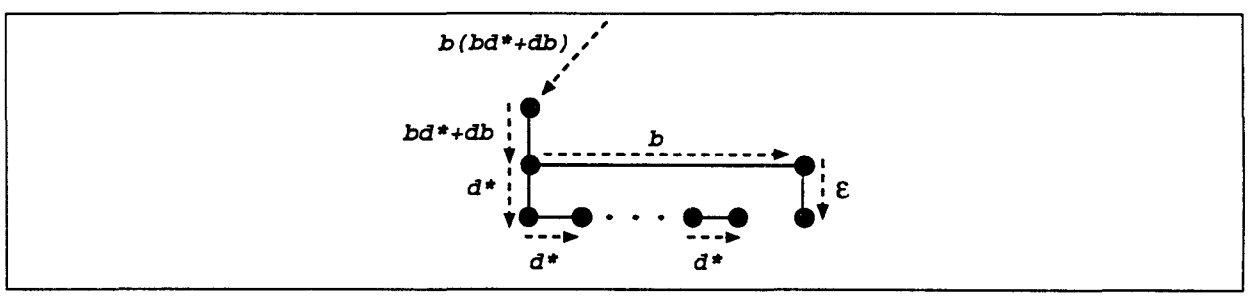



Figure 5.8 : Copie dirigée par la fenêtre : Propagation du descripteur

## 5.7 Ordonnancement

Dans le modèle d'implantation de $P^3$, trois types de threads sont utilisés pour le moment : les threads de traitement, les threads de copie et les threads de régulation de charge. Les threads de traitement exécutent les paquets issus du partitionnement du programme et ceux exécutant les fonctions à parallélisme de données. Les threads de copie permettent de réaliser le mécanisme de copie. Les threads de régulation de charge seront présentés dans la troisième partie de la thèse.

Le problème d'ordonnancement dans notre modèle d'implantation concerne ces trois types de threads. La solution que nous proposons est un ordonnancement à plusieurs classes de priorité. Chaque classe correspond à un type de threads. Les priorités sont attribuées aux différentes classes suivant une graduation exponentielle. Les threads de régulation de charge ont une priorité maximale car, d'une part, ils ne sont pas nombreux, d'autre part, ceci permet de disposer d'informations de charge récentes. La valeur de la priorité des threads de régulation de charge est $p1 = 2^M$, $M$ est un paramètre qui doit être fixé. La priorité des threads de copie est égale à $p2 = 2^{M-1}$. Les threads de traitement n'ont pas la même priorité selon qu'ils déclenchent ou non des copies. Les threads ne déclenchant pas de copies, comme les threads qui exécutent le produit scalaire dans l'exemple du produit matriciel, sont des consommateurs de données. Ils ont la même priorité que les threads de copie i.e. $2^{M-1}$. Par contre, les threads déclenchant des copies ont une priorité *adaptative* car elle dépend de la charge de machine. En effet, si la machine est surchargée (pas de sites en sous-charge) alors la priorité de ce type de threads de traitement est fixée à $p3 = 2^{M-2}$, sinon elle est égale à $2^{M-1}$. Le principe de cette adaptativité est similaire à celui du mécanisme d'étranglement défini dans [RS87].

La gestion des priorités attribuées aux threads dans chaque tâche du modèle d'implantation est assurée par l'ordonnanceur local (MARCEL) de $PM^2$.

## 5.8 Conclusion

Dans ce chapitre, nous avons défini une nouvelle approche d'implantation du modèle $P^3$ sur architectures MIMD sans mémoire commune. Le modèle de programmation supportant l'implantation étant le modèle $PM^2$. L'approche proposée comporte deux tâches principales appelées "P3_START" et "P3_WORKER". La tâche "P3_START" a pour rôle de créer une tâche "P3_WORKER" sur chaque site de la machine puis de lancer l'application. Les tâches "P3_WORKER" s'occupent de l'exécution du programme.

Le modèle d'implantation comprend une partie fixe et une partie variable. La partie fixe contient le code implémentant les deux types de tâches et une librairie de toutes les fonctions prédéfinies du langage Graal implémentées selon le modèle d'évaluation $P^3$. La partie variable est une librairie contenant toutes les fonctions implémentant les différents paquets du programme transformé.

Dans ce chapitre, nous avons également décrit les mécanismes utilisés pour l'exploitation du parallélisme horizontal et du parallélisme vertical, la gestion des copies et l'ordonnancement. Le parallélisme horizontal est exploité en présence des formes fonctionnelles génératrices de parallélisme et des fonctions polyadiques en tenant compte de la taille des fenêtres et de la

charge courante de la machine. Le parallélisme vertical est exploité en présence des fonctions non strictes selon la charge de la machine.

Dans notre modèle, le mécanisme de copie utilise, d'une part, une approche guidée par la notion de fenêtre, ce qui permet son optimisation. D'autre part, il reprend la technique basée sur un seuil de regroupement définie dans [Ben94, BM95]. Ceci permet un réajustement dynamique de la répartition des données impliquant un bon degré de localité et un bon degré de parallélisme. La copie proprement dite de la donnée est complexe car elle dépend de la taille de celle-ci ainsi que de sa répartition courante. En effet, d'une part, une donnée de petite taille et éclatée sur plusieurs sites nécessite un regroupement sur un seul site. D'autre part, une donnée de grande taille et éclatée nécessite une redéfinition de la distribution lors de sa copie. Dans les deux cas, un effort considérable de synchronisation est indispensable.

Par ailleurs, l'approche d'ordonnancement utilisée est une stratégie à plusieurs classes de priorité. Les priorités sont attribuées aux différentes classes de threads suivant une graduation exponentielle. Les threads de traitement générateurs de copies ont une priorité adaptative. Les priorités associées aux différents threads du programme sont gérées par l'ordonnanceur MARCEL de $PM^2$.

# Partie III

# Régulation dynamique de charge

# Chapitre 6

# Régulation de charge : Etat de l'art

## 6.1 Introduction

Le parallélisme est de plus en plus confronté au problème de régulation de charge. En effet, d'une part, la complexité des applications ne cesse d'augmenter depuis quelques années. D'autre part, malgré l'évolution considérable des architectures parallèles et distribuées, le rapport de croissance entre le matériel et le logiciel reste limité. Très souvent, pour des raisons essentiellement économiques, les ressources matérielles disponibles ne répondent pas aux exigences des programmeurs en termes surtout de rapidité d'exécution. Un recours au moyen logiciel est alors nécessaire pour pallier ces limites physiques. Cela consiste en la mise en place d'algorithmes de placement/régulation de charge. Ces algorithmes auront pour rôle d'allouer les ressources de la machine (mémoires, processeurs, ...) aux composants (données, processus, ...) de l'application de façon à répondre le mieux possible aux attentes du programmeur (minimisation du temps d'exécution des programmes, gestion de la tolérance aux pannes, ...).

Les outils d'aide à la programmation d'architectures parallèles et distribuées existants permettent à l'utilisateur de gérer lui-même l'allocation des ressources au moyen de primitives spécifiques. Cette tâche s'avère très fastidieuse. Aussi, un bon nombre de travaux ont été faits pour l'automatiser. Ces travaux sont de deux tendances suivant l'instant où l'allocation est décidée : les méthodes statiques allouent la charge au chargement ou à la compilation du programme ; à l'inverse, les stratégies dynamiques effectuent une allocation à l'exécution. Dans la suite de ce chapitre, nous donnons une taxonomie des algorithmes proposés dans la littérature pour résoudre le problème d'allocation/régulation de charge.

Avant tout, nous donnons les paramètres et les objectifs d'un régulateur de charge. Puis, nous faisons un petit tour d'horizon des méthodes de régulation statique de la charge. Ensuite, nous décrivons les composants d'un système de régulation dynamique de la charge et nous proposons une classification des algorithmes existants selon ces composants. Une étude de cas, en l'ocurrence les langages fonctionnels, est également présentée.

## 6.2 Paramètres et objectifs

La définition d'un algorithme de régulation de charge soulève les points suivants :

- Quels sont les paramètres d'entrée du régulateur de charge (caractéristiques de l'application à réguler et de la machine cible) ?
- Pourquoi réguler la charge ?
- A quel moment de la vie d'une application réguler la charge de celle-ci ?
- Comment réguler la charge ?

Les réponses à ces questions sont détaillées dans les sections suivantes.

### 6.2.1 Paramètres d'un régulateur de charge

Les paramètres qui interviennent dans la définition d'une stratégie de régulation de charge sont essentiellement liés :

- Aux spécificités de l'application ;
- Aux caractéristiques de l'architecture cible de l'exécution de l'application ;

#### 6.2.1.1 Spécificités de l'application

Les paramètres liés à l'application sont, entre autres, les suivants :

1. ***Le type des entités de régulation***

   - **Les fichiers** : Ce type d'entités est utilisé principalement dans le cadre des bases de données réparties. Il convient de trouver un algorithme de placement des différents fichiers constituant une base de données sur les différents sites de stockage de la machine [GS90, BCS89].
   - **Les programmes** : On considère un ensemble de programmes. Chaque programme est une entité atomique pour laquelle il faudra choisir un site d'exécution. La régulation de programmes est abordée avec deux approches différentes. La première approche est celle qui considère de manière globale tous les programmes avec leurs caractéristiques (dates de début d'exécution, durée d'exécution) connues à l'avance [CA82]. La deuxième approche considère chaque programme de façon isolée [HCG+82, HTG96] ; le choix du site d'exécution d'un programme est décidé sur la base de l'état courant du système. Ce type d'approche est plus adapté aux systèmes multi-utilisateurs car pour ces derniers, le nombre de programmes est imprévisible. Une synthèse des méthodes adoptant cette approche est présentée dans [GBS91].

- **Les données** : Les composants concernés par la régulation de charge sont les structures de données de l'application. La répartition de données est utilisée en général pour les applications typiquement à parallélisme de données (ou SIMD : Single Instruction Multiple Data streams) [Fon94]. Mais elle peut également concerner les applications à parallélisme de tâches. On distingue particulièrement la recherche dans un espace de solutions telle que $A^*$ [Nil80] et $IDA^*$ [Kor85] en intelligence artificielle, l'évaluation des programmes suivant le modèle de réduction de graphe parallèle [Jon87] ou le modèle $P^3$ [BM95] dans le cadre de l'évaluation parallèle des langages fonctionnels, le parcours parallèle d'un arbre de résolution dans les langages logiques [CS89], ... Dans chacune de ces applications, il s'agit de placer un graphe ou arbre de données dont la structure évolue dynamiquement dans le temps.
- **Les tâches** : Ces entités sont aussi appelées *processus lourds.* Elles peuvent être des processus UNIX, des tâches Hélios [pPS89], des tâches PVM [GBea94], ... Ce type d'entités est utilisé pour les applications à parallélisme de tâches (à moyen ou gros grain) i.e. MIMD (Multiple Instruction Multiple Data streams) ou SPMD (Single Program Multiple Data streams). Le placement de tâches est le plus abordé dans la littérature [AP88] surtout après l'avènement des environnements de programmation parallèle tels que PVM [GBea94], MPI [for94], ...etc.
- **Les threads** : Appelés aussi *processus légers.* Par opposition aux processus lourds, le contexte d'exécution des threads (compteur ordinal, pile d'exécution, ...) est très petit. Les threads sont créés au sein d'une tâche et partagent l'espace d'adressage de celle-ci ; la migration de threads est de ce fait délicate. Depuis l'apparition des environnements de programmation multithreadée tels que TPVM[FS94], PM$^2$[NM95, Nam97], Athapascan[Chr94], le placement de threads est de plus en plus étudié. La régulation de charge dans les systèmes de processus légers est traitée dans [Gei96].
- **Les objets** : Ce sont des entités regroupant à la fois du code (traitement) et des données. Le problème du placement de ce type d'entités est apparu avec la naissance de la programmation orientée objet parallèle. Il convient de placer l'activité et les données de chacun des objets, appelés *acteurs* dans [Agh86] et CAC (Composants Actifs de Communication) dans [Cou92], sur les sites de la machine [Hem94].

2. ***La granularité des entités de régulation*** : La régulation de charge n'est pas envisagée de la même façon pour les applications à grain de parallélisme fin que pour celles à gros ou moyen grain de parallélisme. Par exemple, pour les applications à grain fin, l'algorithme de régulation a plus tendance à favoriser le regroupement des entités que la distribution de celles-ci pour minimiser le coût de communication. Par contre, la politique de régulation donne avantage plutôt à la distribution des entités qu'à leur regroupement dans le cas d'applications à moyen ou gros grain pour un meilleur degré de parallélisme.

3. ***La forme du graphe définissant les liens entre les entités*** : Cette spécificité laisse apparaître les questions suivantes :

   - Le graphe a t-il une forme régulière ou irrégulière ?
   - La forme du graphe change t-elle pendant l'exécution de l'application ?

   Le critère de régularité a un grand impact sur le découpage de l'application lequel à son tour a des conséquences considérables sur l'efficacité de la régulation. Il est plus facile de

découper une structure régulière (un tableau de données par exemple) que de découper une structure irrégulière (un arbre de données par exemple) en entités de régulation. Le découpage dont il s'agit est celui qui répondrait aux objectifs de la régulation de charge.

La réponse à la deuxième question influe sur le moment du déclenchement de l'algorithme de régulation de charge. En effet, si la forme du graphe ne change pas pendant l'exécution de l'application, une régulation avant le début de l'exécution (statique) est envisageable. Dans le cas contraire, une adaptation dynamique de la régulation est nécessaire. Dans [ABF93], un réajustement dynamique du grain des entités de régulation (tâches) est basé sur une présentation de la forme courante du graphe représentant un programme fonctionnel en exécution.

#### 6.2.1.2 Caractéristiques de l'architecture cible de l'exécution

On ne peut pas étudier le problème de régulation de charge en faisant abstraction des caractéristiques de l'architecture cible du placement de l'application. Ces dernières sont essentiellement les suivantes :

- ***Le modèle de l'architecture*** : Il y a une corrélation entre le modèle de l'architecure et le type des entités de régulation. Par exemple, on ne peut pas parler de placement de tâches ou de programmes sur une architecture de type SIMD. Le placement de données est le plus approprié à ce type d'architecture.
- ***Le diamètre de la machine*** : C'est la longueur maximale du plus court chemin existant entre deux sites quelconques de la machine. Il représente la métrique du critère de localité d'un régulateur de charge, critère sans lequel ce drenier risque de donner de très mauvaises performances [Ban87]. En effet, le gain apporté par le placement d'une entité sur un site très éloigné peut être sérieusement compromis par le surcoût induit par la distance accrue des communications.
- ***Les caractéristiques des sites de la machine*** : On considère qu'un site d'exécution est composé d'un processeur, d'une mémoire locale et d'une interface de communication.
  - La machine est-elle homogène ou hétérogène? : une machine hétérogène nous incite plus à examiner les caractéristiques des entités de placement (coûts d'exécution des tâches, tailles des fichiers, volumes des données, ...).
  - Les capacités de stockage et de calcul des sites : cette caractéristique est importante dans le cas de machines hétérogènes. En effet, on doit éviter des situations où on exécute un programme de quelques instructions sur une machine beaucoup plus puissante que celle sur laquelle on lance la compilation d'une application qui contient des milliers d'instructions et *vice versa.*
- ***Les propriétés du réseau d'interconnexion***
  - La topologie du réseau : ce paramètre influe sur la performance d'un algorithme de régulation de charge. La topologie peut changer dans certaines machines pendant l'exécution d'une application. C'est pour cela que certains algorithmes prennent en compte ce critère.

- La puissance du réseau en termes de débit et de latence : sur une machine équipée d'un réseau à haut débit et une large bande passante, un algorithme de régulation de charge chercherait plus à distribuer i.e. à avoir un bon degré de parallélisme. A l'inverse, sur une machine avec un réseau ayant des caractéristiques opposées on favoriserait plutôt le critère de localité pour minimiser le surcôut de communication.

### 6.2.2 Objectifs d'un régulateur de charge

Les principaux objectifs visés par les algorithmes de régulation de charge sont :

- L'utilisation maximale des ressources. Selon la ressource et l'entité de régulation, cet objectif peut se traduire par la minimisation du nombre de situations :
  - Où un site est inactif pendant que des processus sont en attente d'exécution dans d'autres sites ;
  - Où une création de données ou d'objets doit être différée alors qu'un espace mémoire suffisant pour cette création est disponible sur d'autres sites de stockage ;
  - Où une application est bloquée sur l'accès aux fichiers d'une base de données.
- La tolérance aux pannes. Cet objectif est souvent réalisé par duplication d'entités (de fichiers [HJ84]) ou migration d'entités (de processus [CA83]).
- L'obtention d'un bon degré d'extensibilité. L'extensibilité est l'un des paramètres d'évaluation d'un algorithme de régulation de charge. Il représente un handicap pour les algorithmes centralisés. C'est pourquoi, un recours à une structure hiérarchique de l'algorithme est souvent faite. Dans [Rav95, FR96], la notion de territoire d'exécution est utilisée pour atteindre cet objectif. Un territoire d'exécution est un ensemble de sites dédiés à l'exécution des tâches d'une application donnée dans un contexte multi-applications.
- La minimisation du coût d'exécution de l'application. Cet objectif est souvent formulé par une optimisation d'une fonction objective sous un ensemble de contraintes. La fonction objective peut inclure des coûts d'exécution de processus et des coûts de communication entre processus. Plusieurs fonctions ont été proposées dans la littérature [AP88, MT91, MLT82]. Les contraintes peuvent être liées à l'architecture dans le cas par exemple où celle-ci est hétérogène (vitesse des processeurs, capacité de stockage des mémoires, ...).
- L'amélioration des performances globales du système. Ce but se traduit par la minimisation du temps de réponse moyen du système. C'est le plus couramment visé par les algorithmes de régulation de charge. Il est traduit en général par l'utilisation maximale des processeurs de la machine [NXG86, SS84, LK90, LR92].

Les objectifs cités ci-dessus ne sont pas exclusifs. Certains algorithmes visent plusieurs objectifs à la fois. Dans [TL89] par exemple, trois buts sont recherchés : améliorer la performance du système en induisant un surcoût de calcul le moindre possible, obtenir une extensibilité d'environ quelques centaines de machines et garantir la tolérance aux pannes.

Dans toute la suite, nous considérerons que :

- les entités de régulation sont des processus (légers ou lourds) ;
- la machine cible a une architecture MIMD homogène ;

La réponse à la question "à quel moment de la vie d'une application réguler la charge de celle-ci?" définit deux grandes classes d'algorithmes : les *algorithmes statiques* qui définissent un placement des processus avant l'exécution de l'application considérée et les *algorithmes dynamiques* qui effectuent le placement pendant l'exécution de celle-ci. Les deux sections suivantes décrivent ces deux classes d'algorithmes. Les réponses à la dernière question i.e. "comment réguler la charge?" y sont également étudiées.

## 6.3 Régulation statique de charge

La régulation statique de la charge effectue une allocation des processeurs aux entités de régulation du programme au chargement ou à la compilation de ce dernier en tenant compte des caractéristiques liées au programme (processus, communications entre processus) et à celles de la machine (processeurs, réseau d'interconnexion). Ces caractéristiques sont connues à l'avance i.e. avant le lancement de l'exécution du programme.

Dans la régulation statique, le programme est en général modélisé par un graphe dont les noeuds sont les processus du programme. Chaque noeud peut être pondéré par le temps d'exécution du processus qu'il désigne. Les arcs du graphe représentent les communications entre les processus. Ils portent souvent chacun le coût de la communication qu'ils repésentent. Le graphe peut être orienté pour désigner le sens des communications.

La machine est représentée par un graphe dont la forme dépend de la classe d'architecture à laquelle elle appartient. Les méthodes de régulation définies pour les architectures MIMD sans mémoire commune, sont de deux classes : celles qui considèrent que le réseau d'interconnexion est statique (hypercube [ERS90], linéaire [SE86], grille [MA87]) et celles qui supposent que le réseau est reconfigurable [LS88].

Le problème de la régulation est de trouver un plongement du graphe représentant le programme dans le graphe désignant celui de la machine de façon à répondre le mieux possible au cahier des charges. Le cahier des charges contient un ou plusieurs des objectifs cités dans la section précédente. Plusieurs approches pour résoudre le problème ont été proposées. On distingue principalement la théorie des graphes, la programmation mathématique et l'utilisation d'heuristiques. Le paragraphe suivant présente la première approche.

### 6.3.1 Approches par la théorie des graphes

Les approches par la théorie des graphes modélisent le problème par un graphe regroupant processus et processeurs. Elles tentent de minimiser le coût total de communication inter-processeurs par application des techniques de la théorie des graphes telles que *la recherche d'une coupe minimale (ou flot maximal)* et *la recherche d'un homomorphisme faible.*

La technique de recherche d'une coupe minimale d'un graphe repose sur l'application de

l'algorithme *min-cut*. Dans [Sto77], Stone a décrit un algorithme simple et efficace pour l'affectation d'un graphe de processus à un graphe composé de deux processeurs. Il a utilisé la technique de recherche d'une coupe minimale dans un graphe biparti dont les deux types de noeuds sont les processus et les processeurs. Le problème de l'algorithme obtenu ne garantit pas un équilibre de charge. L'algorithme a été étendu dans [Lo84] à un nombre quelconque de processeurs en imposant des contraintes au problème : le nombre de tâches par processeur ne doit pas dépasser deux et le nombre total de tâches doit être inférieur à deux fois le nombre de processeurs. Sans ces contraintes l'algorithme ne peut pas être efficace car il a été montré dans [MM89] que le problème est NP-complet dès que le nombre de processeurs dépasse trois.

La recherche d'un homomorphisme faible consiste à trouver une fonction de G1 (graphe des tâches) dans G2 (graphe des processeurs) qui optimise une fonction de coût. Il n'existe pas d'algorithme exact qui résoud ce problème. En effet, par exemple le graphe G1 de la figure 6.1 a un cycle de longueur 3 alors que dans G2, la longueur minimum d'un cycle est de 4. Quel que soit le plongement que l'on fait, on ne peut jamais trouver un homomorphisme faible entre G1 et G2. Par ajout de boucle sur chaque noeud de G2(deux processus communicant peuvent être placés sur le même processeur), on peut garantir l'existence d'un homomorphisme faible entre G1 et G2. Dans [Bok81a], le problème est ramené au problème d'isomorphisme de graphes.

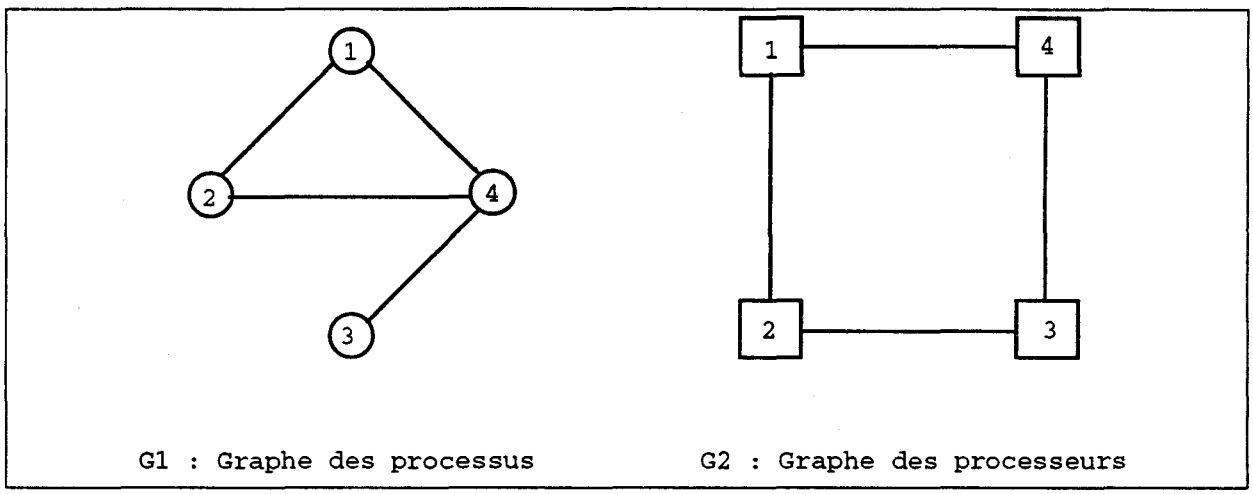



Figure 6.1 : Exemple montrant l'inexistence d'un homomorphisme faible entre G1 et G2

La fonction recherchée est celle qui maximise la cardinalité du placement. La cardinalité d'un placement mesure le nombre de fois que deux processus communicant dans G1 tombent sur deux processeurs liés physiquement dans G2. Dans [ST85], on recherche une fonction qui minimise le coût d'exécution du système par application de l'algorithme A*, utilisé en intelligence artificielle [Nil80].

Les méthodes issues de la théorie des graphes sont très limitatives. Par exemple, l'ajout de contraintes sur le graphe de l'application et celui de la machine rend très complexe l'application de la technique de recherche d'une coupe minimale.
Le paragraphe suivant décrit l'approche par la programmation mathématique.

### 6.3.2 Approches par la programmation mathématique

Les méthodes utilisant la programmation mathématique formulent la question comme un problème d'optimisation combinatoire. Elles utilisent les techniques de programmation mathématique (programmation linéaire, optimisation combinatoire et programmation dynamique) [MLT82, Bok81b].
Le principe de ces méthodes est de ramener le problème à l'optimisation d'une fonction exprimant le coût de la régulation sous un ensemble de contraintes : c'est une énumération "implicite" de toutes les solutions possibles. Un éventail de fonctions qui expriment le coût de la régulation est donné dans [TM90]. Une forme générale de celles-ci est présentée dans [MLT82]. Les contraintes peuvent être des contraintes temporelles, de mémoire, de charge,...etc [TM90].

Parmi les algorithmes basés sur la programmation mathématique, on distingue les algorithmes basés sur la technique "Branch & Bound" (ou BB) [Sin87, ST85] et ceux basés sur la recherche d'un chemin optimal dans un graphe [Bok81b].

Les méthodes BB consistent à parcourir un arbre de décision suivant trois techniques : en profondeur d'abord(BB depth first), en largeur d'abord(BB breadth first) et un parcours hybride(BB best first). Le but du parcours de l'arbre est de rechercher une solution réalisable. Cette technique réduit le nombre d'explorations mais elle reste très énumérative. Pour réduire le plus possible le nombre de noeuds à explorer, on modifie la fonction exprimant le coût de la régulation en introduisant une fonction h dite de "sous-estimation" [Sin87]. La complexité de l'algorithme dépend étroitement de la fonction h.

L'algorithme de Bokhari [Bok81b] est un exemple typique de méthode de recherche d'un chemin optimal. L'algorithme est polynomial pour un nombre quelconque de processeurs mais seulement dans le cas particulier où le graphe représentant le programme parallèle est un arbre, car dans le cas contraire, il a été montré que le problème est NP-complet . Le principe de l'algorithme est le suivant : étant données, d'une part, une application parallèle modélisée par un arbre de processus orienté dans le sens de la création de ces derniers (graphe d'appel), d'autre part, une machine MIMD sans mémoire commune hétérogène, il s'agit, dans un premier temps, de construire un arbre dit "de correspondance", qui représente les différentes affectations possibles des processus aux processeurs. Ensuite, une technique de programmation dynamique est utilisée pour déterminer le chemin optimal qui représente la solution du problème.

Les algorithmes basés sur la théorie des graphes et la programmation mathématique sont des algorithmes exacts. Leur principe repose sur une exploration extensive des solutions possibles ; cela conduit théoriquement à une solution optimale. Cependant, le coût exhorbitant de l'exploration fait que cette approche n'est pas utilisable dans le cas de "grosses applications".

### 6.3.3 Les heuristiques

Dans sa globalité, le problème de régulation est NP-complet. Il est donc illusoire de penser trouver un algorithme qui puisse donner une solution optimale en un temps polynomial. Aussi, a t-on recours aux méthodes approchées ou heuristiques. Celles-ci permettent d'obtenir de bons résultats en un temps raisonnable. Elles permettent parfois de trouver une solution optimale mais peuvent aussi fournir une solution très éloignée de la solution recherchée. Ceci est illustré

par un exemple donné dans [PC84].

Il existe deux classes d'algorithmes heuristiques : les algorithmes **gloutons** et les algorithmes **itératifs**. Les algorithmes gloutons sont initialisés par une solution incomplète qu'ils cherchent à étendre. Un choix établi à une itération n'est jamais remis en cause à l'étape suivante ; chaque choix minimise une fonction partielle de coût. Les algorithmes itératifs sont initialisés par une solution complète qu'ils cherchent à améliorer à chaque itération. A chaque étape de l'algorithme, un nouveau placement est déterminé en effectuant des échanges de processus entre processeurs. Une décision sur la prise en compte du résultat est donnée par un critère de coût. Le placement initial conditionne la rapidité d'un algorithme itératif ; c'est pourquoi, il est nécessaire d'utiliser, dans un algorithme itératif, un algorithme glouton pour déterminer une solution de départ puisque les gloutons sont généralement rapides.

Trois heuristiques parmi celles proposées ont émergé : le groupement de processus, la limitation des routages et le recuit simulé [TM90].
Une approche de la méthode de groupement de processus est décrite dans [MA87]. Elle consiste en deux étapes : la première étape détermine un groupement initial ; la deuxième phase de l'algorithme effectue un équilibrage de la charge des processeurs en effectuant des échanges de processus entre groupes. La contrainte de cet algorithme est qu'il s'intéresse uniquement au cas où le graphe des processeurs est une grille. De bons résultats ont été obtenus seulement dans le cas où le graphe des processus est régulier, planaire et posséde un haut degré de localité.

La limitation des routages cherche à mettre, autant que possible, deux processus communicant dans le programme sur deux processeurs liés physiquement dans la machine. Bokhari [Bok81a] a proposé un algorithme basé sur le principe du "pair-wise exchange". Cet algorithme consiste à effectuer, d'une façon itérative, des permutations sur les colonnes de la matrice associée au graphe modélisant le programme. L'objectif est d'améliorer la cardinalité du plongement. Le problème de cet algorithme est qu'il ne tient pas compte des routages qui peuvent exister entre deux tâches placées sur deux processeurs non liés physiquement. Dans [AP88], André et Pazat ont proposé un algorithme, appelé "Bokhari amélioré", qui résoud ce problème.

La méthode du recuit simulé a été developpée en mécanique statistique pour simuler l'évolution de l'état d'un solide en fonction de la température. Cet état est caractérisé par une énergie E et une disposition des particules composant le solide. En faisant décroître la température, l'énergie de chaque particule décroît jusqu'à atteindre un état d'équilibre qui constitue un minimum local. Pour sortir de ce minimum local, on s'autorise à perturber le système par agitation thermique. Dans le cadre du placement de processus, la configuration du solide correspond à un placement, l'énergie est représentée par le coût du placement et la température est désignée par un paramètre de contrôle c. Le tableau de la figure 6.2 montre l'analogie entre la méthode du recuit simulé et le problème du placement. La perturbation du placement peut se faire par un simple échange de processus.

Deux autres types d'heuristiques ont émergé ces dernières années. Il s'agit des algorithmes génétiques [TB91] et de l'heuristique tabu [Glo89]. L'utilisation de ces heuristiques est plus générale. En effet, celles-ci peuvent être utilisées pour d'autres applications telles que la robotique [BATM94].

Il existe d'autres solutions qui sont mixtes, au sens qu'elles reposent sur une combinaison des trois familles d'algorithmes. Dans [Bou92], l'auteur fait un regroupement des processus en utilisant le recuit simulé puis applique l'algorithme de Bokhari au graphe des groupes de

| Problème de placement de tâches | Méthode du recuit simulé |
|---|---|
| COUT du placement | Energie du solide |
| Affectation des processeurs aux tâches | Configuration du solide |
| Paramètre de contrôle : c | Température : t |

Figure 6.2 : Similitude entre la méthode du recuit simulé et le problème de placement de tâches

processus ainsi obtenu. Le même principe est utilisé dans [Mel92] pour placer un arbre de processus sur un réseau de machines en deux étapes. La première étape de l'algorithme consiste à regrouper les processus suivant une stratégie qui cherche à équilibrer la charge entre processeurs et une autre qui tente plutôt de minimiser le surcoût de communication induit. La deuxième étape applique l'algorithme de Bokhari aux processus agglomérés à la première phase.
Une autre solution mixte est présentée dans [BTV92]. Cette solution utilise un algorithme glouton pour faire un placement initial. Ce placement est amélioré, de façon itérative, en effectuant des relevés de traces. La méthode s'inspire du recuit simulé pour obtenir un minimum global à partir des minimums locaux obtenus [CT93].

Dans cette section, nous avons fait un survol des méthodes d'allocation statique les plus fréquentes mais il existe d'autres approches : les chaînes de Markov, la théorie des files d'attente, la théorie de l'évolution, les réseaux de neurones, les réseaux de Pétri, ...etc. Des références à ces approches sont données dans [TM90].
D'autres synthèses des méthodes de placement statique de processus se trouvent dans [AP88, BCS89, MT91, PPTV91].

## 6.4 Régulation dynamique de charge

La plupart des applications parallèles ont un comportement à l'exécution difficile voire impossible à prévoir. Ainsi, même si les entités de régulation d'une application sont initialement réparties de façon parfaitement équilibrée entre les sites de la machine, cet équilibre est vite remis en cause pendant l'exécution. Aussi, faut-il redistribuer la charge dynamiquement, ce qui revient à définir des algorithmes de régulation de charge qui ne sont plus exécutés ni au chargement, ni à la compilation du programme mais pendant l'exécution de celui-ci. Il s'agit d'algorithmes de **régulation dynamique** de la charge.

A la différence des algorithmes de régulation statique, les méthodes de régulation dynamique prennent des décisions de transfert d'entités et de détermination des sites cibles du transfert en fonction de l'information de charge courante du système (partielle ou globale). Aussi, les tâches que doit réaliser un algorithme de régulation de charge sont principalement :

- La mesure et la collecte de l'information de charge ;

- La prise de décisions de transfert de charge entre les sites de la machine ;
- La détermination des sites cibles des transferts.

### 6.4.1 Composants d'un régulateur dynamique de charge

Les trois fonctions d'un régulateur dynamique de charge ci-dessus sont réalisées chacune par un composant de l'algorithme appelé *agent.* Ainsi, un algorithme de régulation dynamique de charge est constitué de trois agents :

- **Un agent d'information** : Son rôle est de fournir aux deux autres agents les informations concernant la charge du système ;
- **Un agent de transfert** : Cet agent décide, en fonction des informations de charge fournies par l'agent d'information, des transferts de charge entre les différents sites de la machine ;
- **Un agent de localisation** : Celui-ci choisit, dépendamment des informations de charge fournies par l'agent d'information, les sites cibles des transferts décidés par l'agent de transfert.

Si l'efficacité d'un algorithme de régulation statique réside dans l'exactitude de la fonction de coût considérée, celle des méthodes dynamiques dépend de l'efficacité des trois agents ci-dessus.

La suite de ce paragraphe décrit les trois agents et présente une classification des algorithmes existants selon ces agents.

#### 6.4.1.1 L'agent d'information

L'efficacité d'un algorithme de régulation de charge dépend étroitement de la complexité de l'information de charge gérée dans le système. Selon ce critère, on distingue *les algorithmes aveugles* et *les algorithmes intelligents.*

- Les algorithmes aveugles n'utilisent pas d'information de charge par soucis d'induire moins de surcoût de calcul et de communication. Dans cette classe d'algorithmes, deux types de méthodes ont émergé : les *méthodes aléatoires* et les *méthodes cycliques.*
  - Les méthodes aléatoires [ELZ86] désignent au hasard un site candidat en considérant soit toute la machine ou le voisinage seulement du processeur expéditeur.
  - Les méthodes cycliques [WM85] choisissent le site candidat à partir d'une file d'attente qu'elles gèrent cycliquement i.e. avec une politique de type "Round-Robin".
- Les algorithmes intelligents sont basés sur une connaissance partielle ou globale de l'état de charge de la machine. De ce fait, pour cette classe d'algorithmes, l'agent d'information est plus complexe. Ce dernier comporte deux éléments :
  - **Une fonction de caractérisation de la charge locale** d'un site ;

- **Une politique de collecte d'informations de charge** sur les différents sites de la machine. Le but de cette politique est de maintenir un état de charge partiel ou global dans la machine.

**6.4.1.1.1 Caractérisation de la charge locale d'un site** Lors de la définition d'une fonction d'estimation de la charge locale d'un site, trois questions se posent :

- Quels sont les indicateurs de charge qui interviennent dans cette fonction? ;
- Dans le cas où plusieurs indicateurs sont utilisés, quelle est la formule les combinant? ;
- A quels moments appeler cet éstimateur de la charge locale dans un site donné?.

Plusieurs indicateurs de charge sont proposés dans la littérature. Parmi ces indicateurs, les plus couramment utilisés sont :
- Le nombre de processus en attente dans la file du processeur [HJ84, KL84] ;
- Le nombre de processus prêts dans le processeur [MDLT96d] ;
- Le taux d'utilisation de la CPU [HJ84] ;
- Le taux d'occupation de la mémoire [HJ84, KL84] ;
- Le nombre de messages en attente de traitement dans le processeur [BM95].

Le choix des indicateurs de charge à considérer dans la fonction d'estimation dépend des objectifs de l'algorithme de régulation de charge. Par exemple, les trois premiers indicateurs répondent mieux à l'objectif visant à réduire le temps de réponse du système. Le quatrième indicateur est plus intéressant lorsque le but recherché est une gestion efficace de la mémoire globale. Le dernier indicateur est utilisé plus par les algorithmes visant à réduire les communications dans le réseau. Il peut être également utilisé dans un objectif de réduire le temps de réponse du système car il est en bonne corrélation avec la charge future de travail de la machine.

Le nombre d'indicateurs à prendre en compte est un facteur critique. En effet, d'une part, on doit considérer tous les indicateurs de charge pertinents et révélateurs d'une charge exacte et répondant aux motivations de l'algorithme de régulation. Ce qui augmente la complexité temporelle de la fonction de calcul de la charge. D'autre part, ce nombre doit être le plus petit possible pour minimiser le surcoût induit par l'exécution de la fonction de calcul de la charge. Un compromis est alors à trouver entre la pertinence des indicateurs qu'il faut considérer et le surcoût qu'ils induisent dans le calcul de la charge. En général, au plus deux indicateurs sont pris en compte dans la formule de calcul de la charge [LK90, HJ84].

Dans le cas où plusieurs indicateurs sont utilisés, une autre question qui se pose est "comment combiner ces derniers?" (deuxième question ci-dessus). Des coefficients sont généralement introduits pour pondérer les indicateurs dans la formule de calcul. Ces poids sont souvent des paramètres de simulation. Dans [LK90], la fonction d'estimation de la charge d'un site est une combinaison du nombre de processus présents dans le processeur et du taux d'occupation de la mémoire. La formule de calcul est la suivante :

$$chargeLocale = nbProc + \alpha.\frac{1}{1 - tm}$$

où :
chargeLocale : charge locale du site ;

nbProc : nombre de processus dans le site ;
tm : taux d'occupation de la mémoire ;
$\alpha$ : paramètre de simulation fixé à 0.01.

Le nombre d'indicateurs de charge intervenant dans le calcul de la charge d'une machine n'est pas le seul paramètre ayant une influence sur la complexité temporelle de la fonction d'estimation de la charge. En effet, la fréquence de mesure i.e. d'appel à la fonction d'estimation de la charge est également un facteur non négligeable. Cette fréquence est critique car une mesure abusive de la charge permettrait de disposer à tout instant d'une information courante mais induirait un surcoût de calcul important. A l'inverse, une fréquence de mesure trop limitée aurait pour conséquence d'avoir une information locale obsolète mais provoquerait par contre moins de surcoût de calcul. La fréquence d'appel de la fonction d'estimation est, en général, déterminée par la politique de collecte d'informations de charge pour le maintien d'un état de charge partiel ou global dans le système. Le paragraphe suivant présente les différentes fréquences utilisées par les algorithmes de régulation existants.

**6.4.1.1.2 Politique de collecte d'informations de charge** Pour que l'agent de localisation d'un algorithme de régulation puisse prendre des décisions de localisation au sein d'un site donné, il faut qu'il ait connaissance de l'information de charge des autres sites. Cette connaissance peut être globale i.e. elle porte sur toute la machine ou partielle si, au contraire, elle porte juste sur une partie de la machine. Cette exigence de connaître l'état de charge externe à un site traduit la nécessité d'une collecte d'informations pour le maintien d'un état de charge courant dans la machine. Par conséquent, un algorithme de régulation dynamique de charge doit disposer d'une politique de collecte d'informations de charge. La définition d'une telle politique fait apparaître les questions suivantes :

- Quelle type d'information de charge maintenir dans la machine : partielle ou globale? ;
- Où doit être stockée l'information de charge dans la machine? ;
- Quelle est la structure des échanges d'informations de charge? ;
- Quelle est la fréquence de collecte des informations de charge?.

Une information partielle (resp. globale) est une collection d'informations de charge locales des sites d'une partie de la machine (resp. toute la machine). L'information locale d'un site peut être la valeur de la charge locale de celui-ci. Elle peut être également l'état de charge du site. Trois états caractérisent la charge d'un site : *légèrement chargé*, *normalement chargé* et *lourdement chargé*. Très souvent, deux seuils *seuilBas* et *seuilHaut* sont utilisés pour déterminer l'état d'un site de la façon suivante :

- Un site est dit *légèrement chargé* si sa charge locale c vérifie la condition suivante :

$$0 \leq c < seuilBas$$

- Un site est dit *normalement chargé* si sa charge locale c vérifie la condition suivante :

$$seuilBas \leq c < seuilHaut$$

- Un site est dit *lourdement chargé* si sa charge locale c vérifie la condition suivante :

$$seuilHaut \leq c < \infty$$

La réponse à la question "information partielle ou globale?" dépend énormément du diamètre de la machine cible de l'exécution. En effet, la recherche d'une information globale dans une machine à grand diamètre induirait beaucoup de communications, ce qui dégraderait certainement les performances de l'algorithme de régulation ; le risque d'écroulement du système est important. Dans ce cas, une connaissance partielle de l'information est plus intéressante : elle permet une extensibilité nettement meilleure.

Un problème auquel est confronté l'agent d'information lorsqu'il est basé sur une connaissance partielle de l'information de charge est de garantir que la régulation obtenue par l'algorithme est globale. D'autres efforts sont alors nécessaires pour traiter ce problème. Dans [LK90, TB91], une régulation globale est obtenue par propagation successive des échanges locaux d'informations de charge.

Les deux questions concernant le choix du (ou des) site(s) de stockage de l'information de charge et la structure de collecte de celle-ci, sont liées. Nous allons les traiter en même temps. Selon le schéma des échanges des informations, l'agent d'information peut avoir une structure *centralisée*, *hiérarchique* ou *distribuée*.

- Un *agent d'information centralisé* maintient une information globale en effectuant des échanges suivant le schéma maître/esclave. Le maître détient un vecteur dans lequel est stockée la charge locale de chaque site de la machine. Les sites esclaves évaluent chacun leurs charges locales et les remontent au site maître. Dans certains algorithmes, une copie du vecteur de charge globale est diffusée à tous les esclaves. Dans ce cas, les décisions de localisation ne passent pas par le maître. Par contre, dans le cas où cette diffusion n'a pas lieu, l'agent de localisation est centralisé.
  Le problème majeur d'une structure centralisée des échanges est l'extensibilité : les algorithmes centralisés ne sont pas en général extensibles. En effet, quand le diamètre de la machine est grand les remontées des valeurs des charges locales par les esclaves provoquent un goulot d'étranglement au niveau du maître. De ce fait, on a recours au modèle hiérarchique.

- Un *agent d'information hiérarchique* maintient en général une information de charge globale. C'est un modèle centralisé à plusieurs niveaux. Dans ce modèle, les sites sont logiquement regroupés en grappes. Au sein de chaque grappe, il y a un site maître qui maintient un vecteur de charge de tous les sites (esclaves) appartenant à cette grappe. Une information de charge sur la grappe est alors calculée sur la base de ce vecteur. L'information calculée peut être par exemple la charge moyenne de la grappe ou l'état de charge de la grappe (légèrement chargée, normalement chargée ou lourdement chargée). Cette information est ensuite remontée au site maître de niveau supérieur. Le même processus est réitéré à chaque niveau de la structure sauf sur la racine.

- Dans un *agent d'information distribué*, l'information de charge recherchée est en général une information partielle. Chaque site détient la charge de tous les sites appartenant à son voisinage logique (situés à une distance au plus égale à une constante donnée). L'échange

d'information est limité aux sites appartenant à un même voisinage. Dans certains algorithmes, le voisinage d'un site est toute la machine. Un surcoût de communication exhorbitant est dans ce cas induit.

Le choix de la fréquence des échanges d'informations de charge est très critique. En effet, d'une part, un échange très fréquent permet de disposer d'une information globale ou partielle récente à tout instant de l'exécution mais induit un surcoût de communication élevé. A l'inverse, une collecte d'informations moins fréquente provoque moins de surcoût de communication mais l'information disponible est obsolète ; ce qui conduira l'agent de localisation à prendre de mauvaises décisions de placement. Il est donc indispensable de trouver un compromis entre la qualité de l'information de charge collectée et le surcoût de communication que la collecte induit. Quatre protocoles d'échange [Tal95] sont largement utilisés dans la littérature : l'échange continu, l'echange explicite, l'echange relatif et l'echange périodique.

- **Echange continu** : Les informations de charge locale des différents sites sont véhiculées par les messages "utiles" de l'application. Chaque fois qu'un site émet un message, il y introduit sa charge locale. L'avantage de ce type d'échange est qu'il limite les communications dûes à la régulation de charge. Son inconvénient est que, d'une part, il introduit un surcoût de traitement. En effet, à chaque envoi de message, on introduit la charge dans le message. A chaque réception de message, on récupère la charge du site expéditeur. Dans le cas d'applications utilisant beaucoup de communications, le surcoût de traitement est énorme. D'autre part, les sites qui ne communiquent plus ne sont plus informés.

- **Echange explicite** : Avec ce type d'échange, un site désirant connaître la charge d'un autre site fait une demande explicite par envoi de message [SS84, NXG86]. Deux messages sont nécessaires pour avoir l'information. Le surcoût de communication est alors important. De plus, l'information n'est pas fiable étant donné que la charge du site sollicité peut changer pendant la communication.

- **Echange relatif** : Un site communique sa charge lorsque celle-ci a subi une variation significative. Certains algorithmes traduisent cette variation significative de la charge comme une variation impliquant un changement de l'état de charge du site [NXG86]. Dans [HG84], une variation est significative si elle est supérieure à 10%.

- **Echange périodique** : L'échange d'informations se fait périodiquement. Le choix de la période est difficile car on doit trouver un compromis entre le surcoût de communication induit et la qualité de l'information collectée. Une petite période provoque un surcoût moindre tandis qu'une longue période permet de disposer d'une information plus récente. La période peut être fixe [MG95] ou adaptative [XH91]. Une période adaptative est plus intéressante car elle tient compte de l'état de charge courant du système. Cependant, sa gestion pendant l'exécution occasionne un surcoût supplémentaire.

L'information de charge produite par l'agent d'information est utilisée par l'agent de transfert et l'agent de localisation. Le paragraphe suivant décrit l'agent de transfert.

#### 6.4.1.2 L'agent de transfert

L'agent de transfert décide, sur la base de l'information de charge mise à sa disposition par l'agent d'information, des transferts d'entités entre les sites de la machine. L'élaboration d'un agent de transfert soulève les questions suivantes :

- De quel transfert s'agit-il, d'un placement ou d'une migration ?
- Quel site initie le transfert?
- A quel moment la question du transfert est-elle posée ?
- Quels sont les critères de transfert ?

Les algorithmes basés sur le placement décident seulement s'il faut créer les entités localement ou à distance. Une fois qu'une entité est placée sur un site, elle y reste pendant toute sa durée de vie. Ce type de transfert est en général utilisé par les algorithmes recherchant un partage de charge [ELZ86] plutôt qu'un équilibrage de celle-ci. A l'inverse, les algorithmes basés sur la migration peuvent décider à tout moment de l'exécution, de déplacer une entité déjà placée. La migration est en général utilisée par les algorithmes visant un équilibrage de charge [CA82].

Selon la réponse à la deuxième question, deux grandes classes de stratégies de régulation ont émergé : les stratégies *actives* [ELZ86, SS84, XH91] et les stratégies *passives* [NXG86, HTG95]. Les stratégies actives sont celles selon lesquelles les transferts d'entités de régulation sont à l'initiative des sites créateurs de ces entités. Par contre, les stratégies passives sont celles selon lesquelles les transferts sont initiés par les sites en attente de travail.
Le transfert peut être décidé par un seul site, l'agent de transfert est alors centralisé. Il peut être également décidé par plusieurs sites ; dans ce cas, l'agent de transfert a une structure distribuée.

Le mécanisme de déclenchement de l'opération de transfert de charge peut être *aveugle* ou *évènementiel.*

- Le déclenchement de transfert *aveugle* initie le transfert de façon périodique. La période peut être fixe ou adaptative [Dow95].
- Les algorithmes de régulation à déclenchement de transfert *évènementiel* initient le transfert suite à un évènement. La création et la terminaison de processus sont deux exemples d'évènements très souvent utilisés. Le premier est utilisé par les méthodes actives visant indifféremment un équilibrage ou un partage de charge. Par contre, le deuxième évènement est utilisé particulièrement par les méthodes passives et celles visant un équilibrage de charge. Dans [BM95], nous avons utilisé un autre type d'évènement déclencheur du transfert. Il s'agit du mécanisme de copie.

La dernière question sur la politique de transfert concerne les critères de transfert de charge. Il s'agit de déterminer les conditions dans lesquelles un transfert est décidé sur un site. Les conditions peuvent être liées soit aux caractéristiques des entités de régulation ou à la charge des sites de la machine. Le premier type de conditions traduit le problème de granularité : la taille d'une entité sujette au transfert doit être suffisamment importante pour justifier son coût

de transfert. Ce problème a été traité au chapitre précédent.
Suivant le deuxième type de critères, on distingue les algorithmes considérant uniquement la charge locale du site initiateur du transfert des algorithmes prenant en compte les conditions de charge des autres sites. La première classe d'algorithmes utilise en général un seuil de charge. Les méthodes actives (passives) décident d'un transfert dès que la charge d'un site passe au dessus (en dessous) d'un certain seuil. Ce seuil peut être statique ou adaptatif [PTS88].
Certains algorithmes ne se contentent pas de la charge locale d'un site pour décider d'un transfert. En effet, ils considèrent les états de charge des sites de toute (ou une partie de) la machine. Dans [Dow95], quatre déclencheurs ont été définis. Les critères de transfert sont respectivement fondés sur la minimisation de la perte de parallélisme, l'impact et l'efficacité de l'opération d'équilibrage. Une classification des différents mécanismes déclencheurs de régulation (donc de transfert) y est proposée.

Une fois le transfert décidé, un autre problème surgit. Il faut déterminer le site cible du transfert ; c'est le rôle de l'agent de localisation. La présentation de ce dernier est faite dans le paragraphe suivant.

#### 6.4.1.3 L'agent de localisation

L'agent de localisation est chargé de répondre à l'une ou l'autre des deux questions suivantes :

- Vers quel site effectuer le transfert ? Cette question concerne les stratégies de régulation actives ;
- A partir de quel site transférer ? Cette question concerne plutôt les politiques de régulation passives.

La réponse aux deux questions ci-dessus soulève deux autres questions :

- Quelle est la portée maximale des transferts ?
- Quels sont les critères de choix du site destinataire ?

Le critère de portée (ou de localité) d'un transfert influe beaucoup sur le coût des transferts. Selon ce facteur, on distingue les politiques de localisation *locales* et les politiques de localisation *globales*. Les politiques *locales* choisissent le site destinataire du transfert dans un voisinage. Les politiques *globales* considèrent tous les sites de la machine comme candidats cibles des transferts. Dans [KW91], les auteurs montrent que les politiques globales sont efficaces pour la migration de données dans les machines à grand diamètre. Elles donnent de bons résultats pour la migration de processus dans les machines à diamètre moyen. Les politiques locales sont en général les plus utilisées.

Les algorithmes de régulation de charge diffèrent suivant que le choix du site cible du transfert est fait aveuglément ou sur la base de l'information de charge des sites candidats. Le choix aveugle du site cible s'apparente aux méthodes aveugles i.e. aléatoires [ELZ86] ou cycliques [WM85].
La plupart des politiques de localisation basées sur l'utilisation d'une information de charge

désignent le site le moins chargé (cas de méthodes actives) ou le plus chargé (cas de méthodes passives).
Il a été montré dans [XH91] que les stratégies de localisation cycliques donnent de bons résultats pour les machines à grand diamètre. Les politiques désignant le site le moins chargé sont intéressantes plutôt pour les systèmes à diamètre faible.

Pour avoir plus d'informations sur la régulation dynamique de charge, le lecteur peut trouver d'autres synthèses dans [Tal95, BF96].

Dans le paragraphe suivant, nous allons montrer comment le problème de régulation de charge est abordé dans cinq implémentations parallèles des langages fonctionnels, en l'ocurrence MaRS [CDG+86], Flagship [WSWW87], Rediflow [KL84], GRIP [JCSH87] et GUM [THM+96].

### 6.4.2 Etude de cas : Implémentation parallèle des langages fonctionnels

#### 6.4.2.1 MaRS

Dans MaRS, la régulation de charge est dynamique. Elle est particulière : elle est assurée par le réseau de la machine. L'objectif recherché est d'acheminer les messages de création de processus (resp. allocation mémoire) vers les processeurs de réduction ou PRs (resp. les processeurs de mémorisation ou PMs) les moins chargés.

L'agent d'information utilise deux types de charge, la charge mémoire $Cm$ qui représente le nombre de cellules allouées dans le bloc mémoire géré par un PM, et la charge de travail $Cr$ qui est égale au nombre de processus alloués dans un PR. La politique de collecte d'informations de charge utilise un échange relatif. Chaque PR(PM) informe le réseau de communication de sa variation de charge $\Delta Cr(\Delta Cm)$. Ces variations sont propagées jusqu'au premier étage du réseau dans le sens inverse des messages utiles. Chaque processeur de communication (ou PC) dispose de deux ports de sortie PS1 et PS2 sur lesquels il reçoit ces variations. Le problème majeur de cette politique est que si la vitesse de propagation des variations de charge n'est pas suffisamment grande alors il ne sera pas possible d'avoir un état de charge courant du système ; par conséquent, de mauvaises décisions de transfert et de localisation peuvent être prises. Un mécanisme qui pallie ce problème est décrit dans [Cou91].

L'agent de transfert utilise une politique active, la décision de transfert revient à celui qui crée l'entité de régulation (un processus ou une donnée). La question du transfert se pose chaque fois qu'une entité de charge est créée. Il s'agit d'un placement et non pas d'une migration.

L'agent de localisation est entièrement distribué sur les PCs du réseau. La différence de charge d=charge(PS1)-charge(PS2) permet d'orienter les requêtes (de création de processus et d'allocation mémoire) vers le port de sortie le moins chargé. La charge de ce dernier représente la charge de son PC.

#### 6.4.2.2 Flagship

Le problème de régulation de charge dans Flagship concerne tous les paquets exécutables qui composent le programme. L'algorithme de régulation utilisé est dynamique et distribué.

Comme dans la machine MaRS, l'agent d'information est dans le réseau d'interconnexion. L'information de charge maintenue dans le réseau est globale. Chaque processeur maintient deux informations de charge : le niveau d'activité locale NAL (charge locale de travail du processeur), qui représente le nombre de paquets exécutables dans le processeur et le niveau d'activité globale NAG, qui représente le minimum de tous les NAL. Les changements d'état de charge des processeurs sont propagés, dans le réseau, d'étage en étage. Ceci permet de maintenir, dans chaque étage, l'état de charge des processeurs.

La décision de transférer un paquet exécutable d'un processeur vers un autre se fait sur la base des deux informations NAL et NAG de façon distribuée. Si, dans un processeur, NAG est inférieur à NAL alors il est possible d'émettre du travail dans le réseau, sinon on considère que la machine est surchargée, il faut donc éviter la migration du travail.

L'agent de localisation est tel que le travail est orienté de proche en proche en utilisant l'information de charge disponible dans chaque étage du réseau, vers les processeurs légèrement chargés. Pour réduire les communications dans le réseau, on utilise la notion de préférence : à chaque paquet, on associe un processeur préféré, le processeur qui est censé le réécrire. Ceci réduit considérablement le nombre de copies à distance [Sar87].

#### 6.4.2.3 Rediflow

La régulation dans cette machine est dynamique, distribuée et asynchrone. Elle est indépendante de la topologie du réseau. Elle est basée sur le modèle gradient, qui réalise un équilibre global par propagation successive d'équilibrages locaux.

L'agent d'information du modèle gradient calcule une surface de gradient qui lui permet d'assurer la migration de tâches. La surface de gradient représente l'ensemble des proximités des processeurs de la machine, la proximité d'un processeur étant la distance le séparant du processeur oisif le plus proche. Une tâche est conduite par ce gradient vers le processeur en sous charge le plus proche. Chaque processeur maintient un état de sa charge qui peut être "oisif", "normal" ou "surchargé". Un processeur communique son état de charge à ses voisins chaque fois que ce dernier change. Un état de charge est fonction de deux charges, à savoir la charge interne et la charge externe du processeur ; la charge externe est utilisée pour qu'un processeur surchargé ne soit pas envahi par des tâches en provenance des processeurs oisifs. Les formules de calcul de la charge d'un processeur sont données dans [KL84, LK90].

L'agent de transfert utilise une politique de migration mixte i.e. active et passive à la fois. La politique de migration est active car un processeur dépose des tâches dans le réseau dès que sa charge dépasse un seuil donné. La politique est passive car implicitement les processeurs en sous-charge demandent du travail aux processeurs en surcharge. En effet, les tâches déposées dans le réseau sont de proche en proche orientées par les processeurs oisifs par la surface de gradient. De ce fait, l'agent de localisation est totalement distribué.

#### 6.4.2.4 GRIP

Le problème de régulation de charge concerne toutes les tâches créées pour la réduction du graphe associé au programme. Dans GRIP, la régulation est dynamique. L'agent d'information

est centralisé et utilise une information de charge globale. Le site directeur maintient un vecteur de charges des différents sites de la machine et chaque site calcule sa charge locale et la lui transmet. Le vecteur est ensuite diffusé à tous les sites. La charge locale d'un site représente la taille de sa file d'attente de tâches i.e le nombre de tâches dans ce site.

L'agent de transfert utilise la migration. En effet, toutes les tâches sont créées localement. Pour équilibrer la charge de travail dans la machine, certaines tâches migrent des processeurs lourdement chargés vers les processeurs légèrement chargés. Les décisions de transfert sont totalement distribuées. En effet, le transfert est toujours décidé sur la base de la charge locale d'un site. Ces décisions sont basées sur deux seuils, un seuil de sous-charge et un seuil de surcharge. D'une part, un site qui voit sa charge locale dépasser le seuil de surcharge, envoie une partie de sa file de tâches en attente vers une unité de mémoire intelligente (ou UMI). De ce fait, la politique de régulation est active. D'autre part, dès que la charge locale d'un site passe en dessous du seuil de sous-charge, ce site envoie une demande de travail ailleurs. De ce fait, la politique de régulation est passive. La politique de régulation est à la fois passive et active, elle est dite mixte.

L'agent de localisation doit déterminer vers quel site expédier une partie de la file des tâches d'un site en surcharge et à quel site demander du travail en cas de sous-charge d'un site. Le site choisi par l'agent de localisation en cas d'exportation de travail est celui le moins chargé. Les demandes de travail sont soumises aux UMI en commençant par celle se trouvant dans la grappe où se trouve le site. Le surcoût de communication dû aux demandes de travail n'est pas important dans le cas où le système est lourdement chargé, ceci pour deux raisons : premièrement, un site arrive toujours à trouver du travail ; deuxièmement, une UMI peut envoyer plusieurs tâches à la fois. Par contre, si la charge de la machine n'est pas conséquente alors le bus peut être saturé par les réponses négatives aux demandes.

#### 6.4.2.5 GUM

On rappelle que dans GUM, l'exécution d'un programme consiste en une réduction multithreadée du graphe associé. La régulation de charge concerne tous les threads générés pendant l'exécution. Celle-ci est gérée de la manière suivante [THM+96]. Dans GUM, chaque site dispose d'une file locale de threads potentiels (ou FTP). Quand (et seulement quand) un site termine l'exécution de ses threads actifs, il consulte sa file FTP selon une stratégie FIFO. Si la recherche se termine avec succès alors le thread potentiel sélectionné est effectivement créé. Dans le cas contraire, il envoie un message de recherche de travail vers un autre site qu'il aura choisi. On peut donc dire que l'agent d'information n'existe pas.

Un site recevant une requête de travail recherche également avec une stratégie FIFO un thread potentiel dans sa file FTP. Si la recherche réussit alors la demande est satisfaite sinon un autre site doit être choisi et la requête est lui est transmise. Un "âge" est associé au message de demande de travail pour limiter le nombre d'échecs d'une demande. Par ailleurs, en cas d'échec le site demandeur attend un certain délai avant de soumettre à nouveau la demande de travail.

Le site destinataire de la demande de travail est choisi aléatoirement. De ce fait, l'agent de localisation est aveugle. De plus, les transferts sont à l'initiative des sites sans travail. Par conséquent, la politique de transfert est passive.

## 6.5 Conclusion

Lors de l'exécution d'un programme sur une machine parallèle ou distribuée, il est nécessaire de considérer le problème de régulation de charge. La prise en charge manuelle du problème est fastidieuse. De ce fait, un outil automatique i.e. un algorithme doit être developpé. La mise en place d'un algorithme de régulation de charge nécessite avant tout la définition des paramètres du problème et des objectifs de la régulation. Les paramètres peuvent être liés soit aux spécificités de l'application (identification des entités de régulation, granularité des entités, ...) ou à la machine cible de l'exécution (caractéristiques des processeurs et du réseau). Plusieurs objectifs sont poursuivis par les algorithmes de régulation mais le plus largement visé est la minimisation du temps d'exécution des programmes.

La deuxième étape de la mise en place d'un algorithme de régulation est la spécification du fonctionnement de ce dernier. Ceci dépend de l'instant où la régulation est décidée. Selon ce critère, on distingue les méthodes statiques et les méthodes dynamiques. Les méthodes statiques utilisent trois approches principales : la théorie des graphes, la programmation mathématique et les heuristiques. Les deux premiers types d'approches trouvent des solutions exactes mais sont très gourmandes en temps CPU. Par contre, les heuristiques trouvent des solutions approchées voire parfois optimales et en temps polynomial. L'utilisation des algorithmes statiques nécessite la connaissance préalable des caractéristiques du programme (processus et communications). Ce qui n'est pas possible pour la plupart des applications. Dans ce cas, des méthodes dynamiques sont nécessaires car celles-ci tiennent compte de l'état de charge courant de la machine.

Un algorithme de régulation dynamique comporte trois agents : un agent d'information, un agent de transfert et un agent de localisation. L'agent d'information comporte une fonction de caractérisation de la charge locale d'un site et une politique de collecte d'informations de charge des différents sites de la machine. Le choix d'une fonction d'estimation de la charge d'un site nécessite de trouver un compromis entre le coût d'exécution de la fonction et le degré de pertinence de l'information de charge. Au plus deux indicateurs de charge sont en général utilisés. Lors de la définition d'une politique de collecte d'informations de charge, un autre compromis du même type et concernant le choix de la fréquence des échanges d'informations est à trouver. Une fréquence adaptative i.e. qui s'adapte à la variation de charge dans la machine semble être un bon compromis.

L'agent de transfert décide des transferts de charge. Le transfert peut s'agir d'un placement ou d'une migration. Son déclenchement est en général évènementiel (création ou disparition d'une entité de régulation). Les critères de transfert peuvent être liés aux caractéristiques des entités (ex : durée de vie des processus). Ils peuvent également être liés aux conditions de charge de la machine. Ce deuxième type de critères est très souvent considéré. L'agent de transfert utilise en général deux seuils. Le choix des seuils est critique, des seuils adaptatifs sont appréciables.

L'agent de localisation peut être aveugle (ne fait pas appel à l'agent d'information) ou "intelligent"(fait appel à l'agent d'information). Il détermine la source du transfert dans le cas de stratégies passives. Le site choisi est en général le site le plus chargé. L'agent de localisation détermine la destination du transfert dans le cas de stratégies actives. Le site élu est généralement le site le moins chargé. La politique de localisation doit éviter d'inonder les sites destinataires des transferts.

# Chapitre 7

# Une approche de régulation dynamique et adaptative de la charge

## 7.1 Introduction

Nous avons vu dans le chapitre précédent qu'un algorithme de régulation dynamique de charge comporte trois agents : un agent d'information, un agent de transfert et un agent de localisation.

L'agent d'information comporte une fonction de caractérisation de la charge et une politique de collecte d'informations de charge. Lors de la définition d'une telle politique dans les machines à passage de messages, le choix de la fréquence des échanges est critique. En effet, un échange trop fréquent alors que la charge varie très peu (état stable) ne fait qu'occasionner un surcoût de calcul et de communication. A l'inverse, un échange peu fréquent lorsqu'il y a une forte variation de la charge (état fluctuel) a pour conséquence de disposer d'une information obsolète. Notre premier objectif est de trouver une conciliation entre le degré de fraîcheur de l'information et le surcoût induit par sa collecte. Le meilleur compromis semble être l'adaptation de la fréquence des échanges à la variation de la charge.

Les agents de transfert et de localisation doivent prendre des décisions qui ne doivent pas conduire à des inondations de sites. Notre deuxième objectif est donc de mettre en place un mécanisme permettant de prévenir de telles situations.

Afin de minimiser les communications induites par notre algorithme de régulation de charge, une implémentation multithreadée est appréciable. Dans ce but, l'environnement $PM^2$ [NM95, Nam97] est utilisé dans l'implantation de l'algorithme sur une ferme de processeurs DEC/ALPHA.

Dans la suite de ce chapitre, nous montrons de quelle façon se présente le problème de régulation de charge dans notre implantation du modèle $P^3$ (paramètres et objectifs de la régulation de charge). Après, nous décrirons les trois agents de notre algorithme. Ensuite, nous présenterons brièvement l'implantation de l'algorithme et la mesure de certains de ses paramètres sur une ferme de processeurs DEC/ALPHA et sous l'environnement $PM^2$.

## 7.2 Position du problème de régulation de charge dans l'implantatio du modèle $P^3$

Dans le chapitre 5, nous avons identifié trois types de threads dans la nouvelle[1] approche d'implantation du modèle $P^3$. Il s'agit des threads de régulation de charge, de traitement et de copie. Les threads de traitement et les threads de copie sont concernés par la régulation de charge. D'autre part, le placement de ceux-ci est guidé par le placement des copies effectuées pendant l'exécution de l'application. En effet, ils sont exécutés sur les sites où les données qu'ils utilisent sont placées. Par ailleurs, La granularité des threads a été étudiée dans la partie précédente de la thèse. La partie code des threads de traitement est le résultat d'un regroupement effectué sur les fonctions du programme. La donnée du thread est copiée avec un seuil de regroupement. Le contrôle de la granularité des threads de copie est également assurée par utilisation du seuil de regroupement.
Le modèle de machine cible de la régulation est une architecture MIMD sans mémoire commune à diamètre moyen. Les sites de la machine sont homogènes.

L'objectif visé par notre régulateur de charge est la minimisation du temps de réponse des applications. Ce but se traduit par la maximisation de l'utilisation des sites (critère de distribution) et par la minimisation du nombre de communications (critère de localité).

Dans la section suivante, nous décrivons les trois agents composant notre régulateur de charge.

## 7.3 Description de l'approche

### 7.3.1 Agent d'information

L'agent d'information, comme il a été écrit dans le chapitre précédent, comporte une fonction de caractérisation de la charge d'un site et une politique de collecte d'informations de charge des différents sites composant l'architecture. Dans cette section, nous donnons, dans un premier temps, la fonction retenue pour l'estimation de la charge d'un site. Ensuite, nous décrivons la politique de collecte d'informations de charge utilisée pour le maintien d'un état de charge global dans le système. Enfin, nous présentons l'ensemble des primitives et structures de données contribuant à la mise en oeuvre de l'agent d'information.

#### 7.3.1.1 Caractérisation de la charge d'un site

Dans [MDLT96d], pour estimer la charge d'un site, nous avons considéré un contexte mono-application. L'indicateur de charge locale (*icl*) d'un site à un instant donné de l'exécution d'une application est égal au nombre de threads actifs (i.e. prêts pour s'exécuter) de cette application à cet instant.

Dans notre implantation du modèle $P^3$, nous nous plaçons dans un contexte multi-applications. Pour caractériser la charge d'un site, nous considérons, en plus de l'utilisation de la CPU, le taux d'occupation de la mémoire du site. Ceci parce que dans le modèle $P^3$ la quantité de données

[1] Approche différente de celle proposée dans [Ben94]

manipulées est importante vu l'utilisation du mécanisme de copie. La charge mémoire est donc non négligeable. La formule utilisée pour combiner les deux indicateurs de charge est similaire à celle utilisée dans le modèle gradient [KL84]. La charge locale $icl_j(t_i)$ d'un site j à un instant $t_i$ est donnée par :

$$icl_j(t_i) = \tau^j_{cpu}(t_i) + c.(\frac{1}{1 - \tau^j_{mem}(t_i)})$$

Où $\tau^j_{cpu}(t_i)$ et $\tau^j_{mem}(t_i)$ représentent respectivement le taux d'utilisation de la CPU et le taux d'occupation de la mémoire du site j à l'instant $t_i$, et $c$ est une constante fixée à 0.1.
Cette formule permet de minimiser la contribution de la mémoire au calcul de la charge locale d'un site jusqu'à ce qu'elle devienne importante.

#### 7.3.1.2 Politique adaptative de collecte d'informations de charge

Avant de décrire la politique de collecte d'informations utilisée pour le maintien d'un état de charge global dans la machine, nous donnons quelques définitions utilisées par la suite.

##### 7.3.1.2.1 Informations de charge utilisées et notations

**Charge locale d'un site**

**icl**$_j(t_i)$ : Indicateur de charge locale du site j à l'instant $t_i$. La charge locale d'un site a été donnée dans le paragraphe précédent.

**Charge globale de la machine**

**icg**$(t_i)$ : Index de charge globale de la machine à l'instant $t_i$. C'est une table qui contient les indicateurs de charge locale de tous les sites de la machine à l'instant $t_i$. On peut donc écrire :

$$icg(t_i) = (icl_1(t_i), icl_2(t_i), ..., icl_n(t_i))$$

Où $n$ est le nombre de sites dans la machine.

**Charge moyenne de la machine**

**chargeMoy**$(t_i)$ : Charge moyenne dans la machine à l'instant $t_i$. Elle est calculée à partir de la table $icg(t_i)$. La formule de calcul est la suivante :

$$chargeMoy(t_i) = \frac{\sum_{j=1}^{n} icg[j](t_i)}{n}$$

Où $n$ est le nombre de sites dans la machine.

### Etat de charge global de la machine

**iecg($t_i$)** : Index d'état de charge globale de la machine. C'est la collection de tous les sites légèrement chargés à l'instant $t_i$. Si $k$ est le nombre de sites légèrement chargés à l'instant $t_i$ alors on peut écrire :

$$iecg(t_i) = (S_1, S_2, ..., S_k)$$

La définition d'un site légèrement chargé est donnée ci-dessous.

### Etats de charge d'un site

La plupart des systèmes de régulation de charge utilisent deux seuils de charge souvent statiques pour définir l'état de charge d'un site. Dans notre approche, ces deux seuils sont adaptatifs car ils sont fonctions de la charge moyenne de la machine qui est elle-même sensible à la variation de la charge globale. Les deux seuils sont donnés à l'instant $t_i$ par $B(t_i)$ et $H(t_i)$ $(B(t_i) < H(t_i))$. Les formules de calcul des deux seuils sont celles utilisées dans [XH91] et sont les suivantes :

$$B(t_i) = \lfloor (1-\gamma).chargeMoy(t_i) \rfloor \tag{7.1}$$

$$H(t_i) = \lceil (1+\gamma).chargeMoy(t_i) \rceil \tag{7.2}$$

Où $\gamma$ est un paramètre choisi dans l'intervalle $[0, 0.2]$.

Trois états caractérisent la charge d'un site. En effet, celui-ci peut être *légèrement chargé*, *normalement chargé* ou *lourdement chargé*. Les définitions de ces différents états sont données ci-dessous. Dans ces définitions, $c_j(t_i)$ désigne la charge locale d'un site donné j à l'instant $t_i$ de l'exécution. Sa valeur est donnée par :

$$c_j(t_i) = \begin{cases} icg[j](t_i) & si \ la \ charge \ locale \ d'un \ site \ est \ lue \ a \ partir \ de \ icg(t_i) \\ icl_j(t_i) & si \ la \ charge \ locale \ d'un \ site \ est \ calculee \ localement. \end{cases}$$

**Définition 6.1** : Un site j est dit **légèrement chargé** à un instant $t_i$ de l'exécution d'une application donnée si la condition suivante est vérifiée :

$$0 \leq c_j(t_i) < B(t_i)$$

**Définition 6.2** : Un site j est dit **normalement chargé** à un instant $t_i$ de l'exécution d'une application donnée si la condition suivante est vérifiée :

$$B(t_i) \leq c_j(t_i) < H(t_i)$$

**Définition 6.3** : Un site j est dit **lourdement chargé** à un instant $t_i$ de l'exécution d'une application donnée si la condition suivante est vérifiée :

$$H(t_i) \leq c_j(t_i) < \infty$$

En considérant ces définitions des états de charge d'un site, on risque d'avoir des situations où un site ayant une petite charge est déclaré lourdement chargé (cas où la charge moyenne est faible), ou un site de charge importante est déclaré légèrement chargé (cas où la charge moyenne est importante). Ceci peut conduire à une instabilité dans le système car d'une part, une charge supplémentaire peut être envoyée à un site alors que celui-ci est surchargé. D'autre part, on peut se priver d'envoyer de la charge à un site alors celui-ci est sous-chargé. Par conséquent, il est nécessaire d'introduire deux autres seuils *Bmin* et *Hmax* qui représenteront les bornes inférieure et supérieure respectivement des seuils $B(t_i)$ et $H(t_i)$. Autrement dit, *Bmin* (resp. *Hmax*) désigne la valeur du seuil $B(t_i)$ (resp. $H(t_i)$) au dessous (resp. au dessus) de laquelle on considère *Bmin* (resp. *Hmax*) au lieu de $B(t_i)$ (resp. $H(t_i)$) dans les définitions des états de charge des sites. Les définitions précédentes deviennent alors :

**Définition 6.1'** : Un site j est dit **légèrement chargé** à un instant $t_i$ de l'exécution d'une application donnée si la condition suivante est vérifiée :

$$0 \leq c_j(t_i) < Max(Bmin, B(t_i))$$

**Définition 6.2'** : Un site j est dit **normalement chargé** à un instant $t_i$ de l'exécution d'une application donnée si la condition suivante est vérifiée :

$$Max(Bmin, B(t_i)) < c_j(t_i) \leq Min(Hmax, H(t_i))$$

**Définition 6.3'** : Un site j est dit **lourdement chargé** à un instant $t_i$ de l'exécution d'une application donnée si la condition suivante est vérifiée :

$$Min(Hmax, H(t_i)) \leq icg(t_i) < \infty$$

**Structure générale de la politique** La politique de collecte des informations de charge a une structure centralisée[2](figure 7.1). Chaque site esclave dispose d'un processus appelé *S_PROCESS* qui évalue sa charge locale (*icl*) et la remonte au site maître. Ce dernier (site 0 sur la figure 7.1) maintient un index de charge globale (icg) constitué des indicateurs de charge locale de tous les sites esclaves. Cet index est utilisé par un processus appelé *M_PROCESS* (se trouvant sur le site maître) pour diffuser une information de charge globale à tous les sites esclaves. Suivant l'application, cette information peut être le vecteur icg lui-même. Elle peut être également une information calculée sur la base de icg : la charge moyenne des sites, un vecteur de tous les sites légèrement chargés, lourdement chargés ou les deux à la fois. Dans [MDLT96f], l'information diffusée est un vecteur de sites légèrement chargés. Dans l'approche proposée dans cette thèse, il y a une légère différence. En effet, deux informations sont diffusées en même temps : la charge moyenne *chargeMoy* et le vecteur *iecg* des sites légèrement chargés.

Dans le paragraphe suivant, nous montrerons à quelles fréquences les informations de charge sont calculées et collectées dans l'approche que nous proposons.

[2]De ce fait, l'agent d'information est centralisé.

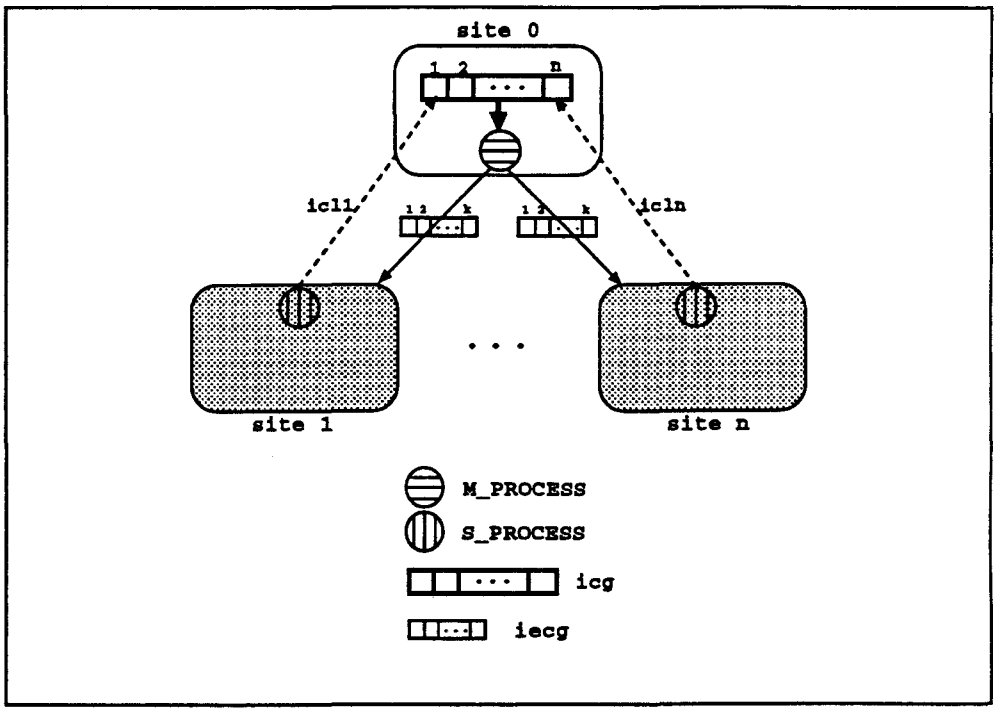



Figure 7.1 : Agent d'information du régulateur de charge

**Fréquence de collecte des informations de charge** Un problème fondamental posé par la définition d'une politique d'information est en général le choix de la fréquence des échanges des informations de charge. Dans notre approche, en particulier, la question posée est la suivante : à quelles fréquences faut-il que :

- Le processus *S_PROCESS* calcule et remonte au maître l'indicateur de charge locale *icl* ;
- Le processus *M_PROCESS* consulte et diffuse l'information de charge globale (*chargeMoy* et *iecg*)?

Le choix de ces fréquences est critique. En effet, d'une part, une remontée trop fréquente des *icl* aurait pour avantage d'avoir un état de charge courant du système mais induirait à la fois un surcoût de calcul et un surcoût de communication. Le surcoût de calcul est dû à l'appel de l'estimateur de *icl* et à la mise à jour de icg. Le surcoût de communication est occasionné par le message porteur de l'*icl*. A l'inverse, une remontée peu fréquente de *icl* aurait pour inconvénient d'avoir à tout instant une information de charge obsolète. Elle provoquerait par contre un moindre surcoût de calcul et de communication. D'autre part, une diffusion excessive de *chargeMoy* et de *iecg* permettrait de disposer d'un état de charge global courant dans chaque site, ce qui permettrait à l'agent de localisation de répartir la charge plus équitablement dans la machine. Toutefois, cet avantage est compromis par le surcoût occasionné. Inversement, une diffusion exagérement limitée de *chargeMoy* et de *iecg* induirait un surcoût moindre mais impliquerait de disposer à tout instant d'une information périmée.

Notre objectif est de trouver des fréquences qui permettraient de maintenir un état de charge

le plus courant possible dans le système et qui, dans le même temps, induiraient un surcoût de calcul et de communication le moindre possible.

Le choix des fréquences que nous avons adopté dans la définition de notre politique de collecte d'informations est le suivant. D'une part, le calcul de *icl* par *S_PROCESS* et la consultation de icg par *M_PROCESS* sont faits *périodiquement.* Les périodes sont appelées respectivement *L_DELAY* et *G_DELAY.* Elles sont *adaptatives* car elles sont dynamiquement réajustées en fonction de la variation respectivement de la charge locale dans chaque site et de la charge globale dans la machine. D'autre part, la communication des informations de charge locale et globale est *relative.* En effet, d'une part, un site esclave remonte sa charge locale *icl* au site maître si et seulement si celle-ci a changé de façon significative par rapport à la dernière fois où elle a été calculée. Dans [MDLT96f], cette variation de *icl* est significative si elle est égale au moins à un seuil $T_{min} = 2$. D'autre part, le site maître diffuse aux esclaves la charge moyenne et le vecteur des sites légèrement chargés uniquement s'il existe au moins un site dont la charge locale (lue à partir de icg) a varié en comparaison de sa valeur à la dernière consultation de icg. Le paragraphe suivant présente la formulation mathématique des périodes (ou délais) *L_DELAY* et *G_DELAY.*

**Calcul des délais L_DELAY et G_DELAY** Avant de présenter les formules de calcul, nous introduisons deux concepts appelés : *facteur $r_l$ de variation locale de la charge* et *facteur $r_g$ de variation globale de la charge.* Ces deux facteurs appartiennent à l'intervalle $[0, 1]$.

**Définition 6.4** : Soient $t_i$ et $t_{i+1}$ deux instants successifs de mesure (éventuellement de remontée) de l'indicateur de charge locale d'un site j. Le facteur $r_l^j(t_i)$ de variation locale de la charge dans le site j entre $t_i$ et $t_{i+1}$ s'exprime par la formule suivante :

$$r_l^j(t_{i+1}) = \frac{|icl_j(t_{i+1}) - icl_j(t_i)|}{Max(icl_j(t_i), icl_j(t_{i+1}))}$$

**Définition 6.5** : Soient $t_i$ et $t_{i+1}$ deux instants successifs de consultation par *M_PROCESS* de l'information *icg*. Le facteur $r_g(t_{i+1})$ de variation globale de la charge dans une machine de $n$ sites entre $t_i$ et $t_{i+1}$ est formulé par :

$$r_g(t_{i+1}) = \frac{1}{n} \sum_{j=1}^{n} \frac{|icg[j](t_{i+1}) - icg[j](t_i)|}{Max(icg[j](t_i), icg[j](t_{i+1}))} \tag{7.3}$$

Les délais *L_DELAY* et *G_DELAY* sont fonctions des deux facteurs de variation de charge $r_l$ et $r_g$. C'est pourquoi ils sont adaptatifs. L'intérêt de $r_l$ et $r_g$ est de révéler les variations de charge dans la machine. Si le facteur de variation locale (resp. globale) de la charge d'un site donné est nul alors il n'y a pas de variation de charge dans ce site (resp. la machine). A l'inverse, si le facteur $r_l$ (resp. $r_g$) est non nul alors il mesure le taux de variation de la charge locale (resp. globale).

Les formules de calcul des délais *L_DELAY* et *G_DELAY* sont données ci-dessous, elles sont récurrentes. Dans ces formules, $t_0$ représente l'instant initial de lancement du système de régulation de charge, $t_i$ et $t_{i+1}$ sont deux instants successifs de mesure de la charge locale dans le calcul

de *L_DELAY*, et de consultation de *icg* dans le calcul de *G_DELAY*. Les paramètres $r_1$ et $r_2$ sont deux constantes fixées, de façon empirique dans [MDLT96d], respectivement égales à *0.01* et *0.1*. *INIT_DELAY* est un paramètre initial dont le calcul sera présenté dans le paragraphe 7.4.2.

**a) Expression de L_DELAY**

- $L\_DELAY(t_0)$ = INIT_DELAY

- $$L\_DELAY(t_{i+1}) = \begin{cases} (1 - r_l^j(t_{i+1})).L\_DELAY(t_i) & r_1 \leq r_l^j(t_{i+1}) \leq r_2 \\ (1 - r_2).L\_DELAY(t_i) & r_l^j(t_{i+1}) > r_2 \\ (1 + r_1).L\_DELAY(t_i) & r_l^j(t_{i+1}) < r_1 \end{cases}$$

**b) Expression de G_DELAY**

- $G\_DELAY(t_0)$ = INIT_DELAY

- $$G\_DELAY(t_{i+1}) = \begin{cases} (1 - r_g(t_{i+1})).G\_DELAY(t_i) & r_1 \leq r_g(t_{i+1}) \leq r_2 \\ (1 - r_2).G\_DELAY(t_i) & r_g(t_{i+1}) > r_2 \\ (1 + r_1).G\_DELAY(t_i) & r_g(t_{i+1}) < r_1 \end{cases}$$

Dans les formules ci-dessus, l'intérêt des paramètres $r_1$ et $r_2$ est de contrôler la fréquence de mise à jour des délais. En effet, quand le système est stable ($r_l, r_g \rightarrow 0$) on augmente le délai (L_DELAY ou G_DELAY) de $r_1$ fois sa valeur, ce qui permet d'éviter un échange infructueux d'information. La valeur de $r_1$ est faible car cela permet de réagir le plus rapidement possible en cas d'un nouveau passage du système à l'état fluctuel. A l'inverse, lorsque la charge varie de manière significative dans le système, on augmente la fréquence des échanges d'informations, ce qui assure le maintien d'un état de charge récent dans le système. Rappelons que le surcoût induit par cet accroissement de la fréquence (et donc du nombre de messages) doit être le moindre possible. De ce fait, on ne doit pas se laisser tenter par l'acquisition d'une information de charge exacte dans le système. Le paramètre $r_2$ est utilisé à cet effet, celui-ci limite la vitesse de décroissance des délais (ou de croissance de la fréquence de collecte de l'information de charge) quand celle-ci devient importante.

Outre le problème du surcoût provoqué par la décroissance des délais, un autre problème se pose lors d'une forte fluctuation de l'état du système. Effectivement, dès que les délais commencent à prendre des valeurs très proches de zéro, il y a une importante accumulation de messages d'information de charge dans les sites et plus particulièrement dans celui qui contient *M_PROCESS* (goulot d'étranglement). Ce phénomène, que nous avons pu observer sur l'application IDA* appliqué au problème du taquin 15 [MDLT96d] exécutée sur une ferme de processeurs DEC/ALPHA, conduit à un écroulement du système. Ce constat nous a amené à minorer la suite des délais (qui est géométrique décroissante positive lorsque le système est fluctuel) par une valeur *LOW_DELAY*. Le calcul de ce paramètre sera présenté dans le paragraphe 7.4.2.

**7.3.1.2.2 Comparaison avec un autre travail** Dans [XH91], les auteurs ont défini des méthodes heuristiques de régulation dynamique de la charge. Ces méthodes intègrent une politique d'information dont la structure est centralisée. Le site maître diffuse périodiquement aux sites esclaves une information de charge globale. La formule exprimant la période de diffusion est similaire à celle exprimant *G_DELAY* avec une différence dans le calcul du facteur $r_g$ de variation globale de la charge dans la machine. Dans [XH91], le facteur $r_g$ est calculé par la formule suivante :

$$r_g(t_{i+1}) = \frac{|\sigma(icg(t_{i+1})) - \sigma(icg(t_i))|}{Max(\sigma(icg(t_{i+1})), \sigma(icg(t_i)))} \tag{7.4}$$

Les quantités $\sigma(icg(t_i))$ et $\sigma(icg(t_{i+1}))$ mesurent les dispersions des charges locales des différents sites respectivement aux instants $t_i$ et $t_{i+1}$. Soient $n$ le nombre de sites dans la machine et $m(t_i)$ la charge moyenne des sites à l'instant $t_i$. La dispersion $\sigma(icg(t_i))$ des charges locales des sites à l'instant $t_i$ est mesurée de la façon suivante :

- $\sigma(icg(t_i)) = \frac{1}{n}\sum_{j=1}^{n}(icg[j](t_i) - m(t_i))^2$
- $icg(t_i) = (icg[1](t_i), icg[2](t_i), ..., icg[n](t_i))$
- $m(t_i) = \frac{1}{n}\sum_{j=1}^{n} icg[j](t_i)$

Comparons les deux formules (7.3) et (7.4) sur un exemple. Considérons $n = 2$ (les sites sont numérotés par 1 et 2) et deux instants de diffusion successifs $t_i$ et $t_{i+1}$. Supposons aussi que :

- A l'instant $t_i$ : $icg[1](t_i) = 1$ et $icg[2](t_i) = 3$. Ce qui donne $m(t_i) = 2$.
- A l'instant $t_{i+1}$ : $icg[1](t_{i+1}) = 3$ et $icg[2](t_{i+1}) = 1$. Ce qui donne $m(t_{i+1}) = 2$.

La formule (7.4) donnera $r_g = 0$ car : $\sigma(icg(t_i)) = \sigma(icg(t_{i+1})) = 0$.

Dans [XH91], un site j est considéré légèrement chargé à un instant $t_i$ si la condition suivante est vérifiée :

$$icg[j](t_i) < (1 + \alpha).\frac{\sum_{j=1}^{n} icg[j](t_i)}{n}$$

$\alpha$ est un paramètre choisi dans [0,0.2].

Selon la condition ci-dessus, les sites 1 et 2 sont respectivement légèrement chargé et lourdement chargé à l'instant $t_i$ mais vice-versa à l'instant $t_{i+1}$. Malheureusement, cette information n'est pas déterminée par l'agent d'information de [XH91], ce qui conduira l'agent de placement à prendre des décisions à effet inverse de l'objectif de la régulation de charge.
Ce genre de problème ne se pose pas en utilisant notre formule i.e. (7.3). En effet, la valeur de $r_g$ obtenue avec la formule (7.3) est $r_g = \frac{2}{3}$, ce qui révèle la fluctuation de l'état global de la charge dans le système.

Nous pouvons conclure que notre formule de calcul du facteur de variation globale de la charge est plus fiable que la formule utilisée dans [XH91] en particulier pour l'exemple présenté. Nous pouvons généraliser cet avantage à toutes les situations similaires à celle illustrée par l'exemple.

Ce qui veut dire toutes les situations dans lesquelles, entre deux instants de diffusion successifs, il y a permutation des valeurs des *icl* des sites, entraînant ainsi un changement d'état de ces derniers. Certes, dans de telles conditions, il n' y a pas de variation de la charge globale, cependant, il y a fluctuation de l'état global de la charge dans la machine. Un bon régulateur de charge est plus sensible à une variation d'état de charge qu'à une variation de la charge elle-même (celle-ci peut ne pas entraîner un changement de l'état de charge).

#### 7.3.1.3 Structures de données et primitives de l'agent d'information

Comme il a été indiqué précédemment, l'agent d'information est constitué de deux types de processus permanents [3] *M_PROCESS* et *S_PROCESS*. Ces deux processus font appel, parfois de façon concurrente, à cinq primitives (décrites ci-dessous). Ces primitives, à leur tour, manipulent quatre structures de données (figure 7.2).

**A) Les structures de données de l'agent d'information**

- ***icg*** (index de charge globale) : C'est une table créée dans le site maître i.e. celui qui contient *M_PROCESS*. Nous rappelons qu'elle contient les indicateurs de charge locale de tous les sites de la machine.

$$icg(t_i) = (icg[1](t_i), icg[2](t_i), ..., icg[n](t_i))$$

- **ancienIcg** (ancien index de charge globale) : C'est le contenu de icg à la dernière diffusion ou tentative de diffusion de l'*iecg* par *M_PROCESS*[4]. Il sert au calcul du facteur $r_g$ de variation globale de la charge, il représente $icg(t_i)$ dans la formule (7.3).

- ***iecg*** (index d'état de charge globale) : C'est la table de sites légèrement chargés diffusée par *M_PROCESS* aux différents sites.

- **iecgLocal** (index d'état de charge globale local) : C'est la version locale de *iecg* diffusé par *M_PROCESS*. Cette table est utilisée par l'agent de localisation pour prendre des décisions de placement.

**B) Les primitives de l'agent d'information**

- ***diffusion_iecg*** : C'est la primitive exécutée par le processus *M_PROCESS*. De ce fait, sa durée de vie est égale à celle de l'application. Son rôle est de calculer et éventuellement diffuser l'*iecg* tous les intervalles de temps G_DELAY.

- ***maj_icg*** : Cette primitive s'exécute sur le site maître. Elle est invoquée à distance par le processus *S_PROCESS* pour mettre à jour la table *icg* par la valeur de l'*icl* de son site.

[3] présents pendant toute la durée de l'exécution de l'application

[4] ancienIcg($t_{i+1}$)=icg($t_i$).

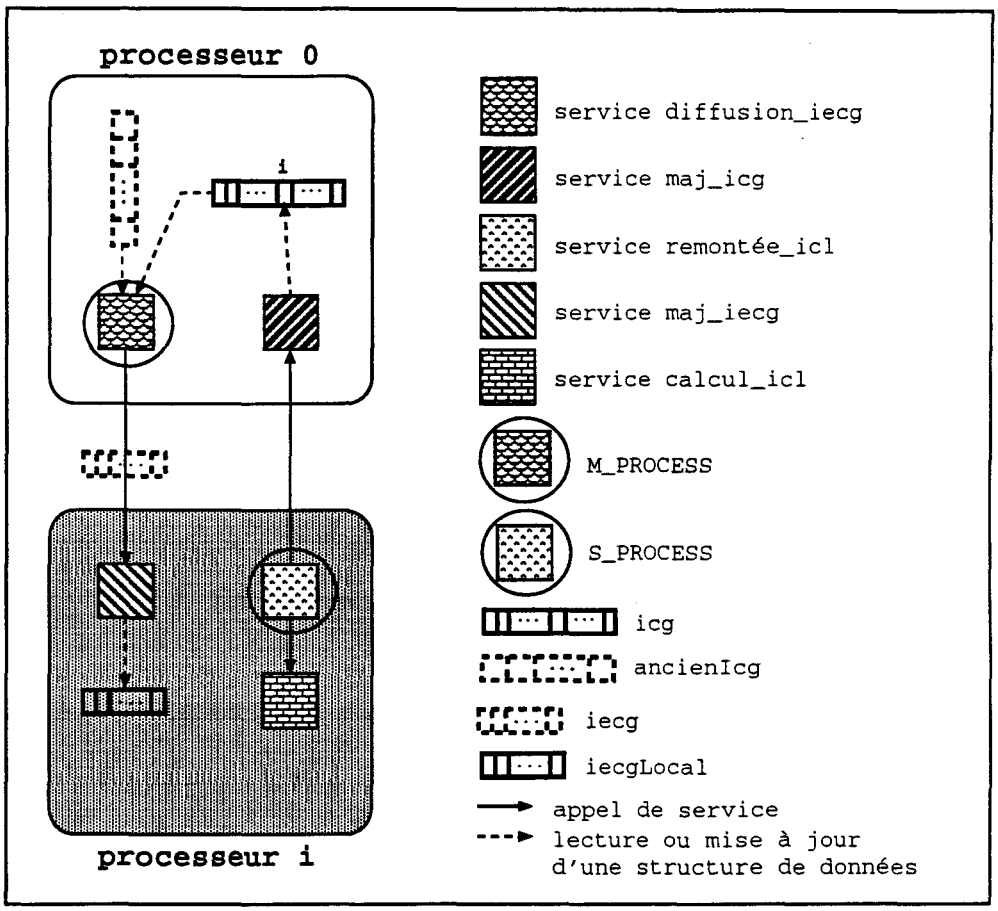



Figure 7.2 : Structures de données et primitives de l'agent d'information

- ***maj_iecg_local*** : C'est une primitive appelée à distance par le processus *M_PROCESS* pour la réception de l'information diffusé. Elle met à jour la table *iecgLocal* et de la charge moyenne *chargeMoy*.

- ***remontée_icl*** : C'est la primitive exécutée par le processus *S_PROCESS*. Par conséquent, comme c'est le cas pour la primitive *diffusion_iecg*, sa durée de vie est égale à celle de l'application. Elle se charge de remonter l'*icl*, calculé par la fonction *calcul_icl* (décrite ci-dessous), au site maître pour mettre à jour la table *icg* par appel distant de la primitive *maj_icg*.

- ***calcul_icl*** : C'est la fonction d'estimation de la charge locale d'un site. Cette primitive est appelée par la primitive *remontée_icl* tous les intervalles de temps L_DELAY.

Dans ce qui va suivre, nous donnerons une description algorithmique des cinq primitives ci-dessus. Nous montrerons également comment ces primitives agissent sur les structures de données de l'agent d'information.

## C) Description algorithmique des différents primitives

Dans ce paragraphe, P et V désignent les primitives de synchronisation classiques de E.W. Dijkstra.

### a) La primitive diffusion_iecg

Dans l'algorithme ci-dessous, la variable *nbProc* désigne le nombre de sites de la machine. *icg* et *ancienIcg* représentent respectivement la valeur actuelle de l'index de charge globale et celle calculée G_DELAY unités de temps auparavant. *variation* est une variable booléenne qui sert de témoin de la variation globale de la charge dans le système [5]. *mutexIcgVar* est un sémaphore d'exclusion mutuelle sur les deux variables *icg* et *variation* partagées par les deux primitives *maj_icg* et *diffusion_iecg*. La procédure *calcul_iecg* calcule l'*iecg* de la machine suivant les critères de surcharge ou sous-charge d'un site définis dans le paragraphe 7.3.1.2.1. La diffusion de l'*iecg* calculé est assurée par la procédure *MULTI_RPC*. Cette dernière a trois arguments : la variable globale *procs* qui contient la liste des sites cibles de la diffusion, la primitive qui reçoit le vecteur diffusé (*maj_iecg_local*) et *iecg* qui contient l'*iecg* à diffuser. La procédure *calcul_G_DELAY* détermine la nouvelle valeur de G_DELAY en utilisant la formule de calcul donnée précédemment.

***Primitive diffusion_icg()***

```
   entier i;
Début
   /* initialisation */
  • Pour i ← 0 à nbSites faire
       icg[i] ← ancienIcg[i] ← 0;
  •G_DELAY ← INIT_DELAY[6];
  • Tant que (VRAI) Début
    • delai(G_DELAY); /* blocage pendant G_DELAY */
    • P(mutexIcgVar);
    • Si (Non variation) alors Début
      •G_DELAY ← (1+r1)×G_DELAY;
      • V(mutexIcgVar);
    Fin Si
    Sinon Début
        /* calcul de iecg */
        • variation ← FAUX;
        • chargeMoy ← moyenne(icg);
        • calcul_iecg(iecg);
        • V(mutexIcgVar);
        /* diffusion de iecg */
        • MULTI_RPC(procs,maj_iecg_local,iecg,chargeMoy);
        • calcul_G_DELAY();
    Fin sinon
```

[5] Nous rappelons qu'il y a diffusion de *iecg* seulement s'il y a variation globale de la charge i.e. s'il y a au moins une variation locale.

[6] INIT_DELAY=G_DELAY($t_0$) donné dans la formule (7.6)

Fin Tant que.
Fin.

### b) La primitive maj_icg

Dans l'algorithme ci-dessous, *icl* représente l'index de charge locale du site de numéro *numsite*. Les variables *variation* et *mutexIcgVar* sont les mêmes que celles utilisées dans la primitive précédente.

***Primitive maj_icg (numsite,icl)***
Début
- P(mutexIcgVar);
- icg[numsite] ← *icl*;
- variation ← VRAI;
- V(mutexIcgVar);

Fin.

### c) La primitive maj_*iecg*_local

Dans cette primitive, *mutexIecgLocal* est un sémaphore d'exclusion mutuelle sur la variable *iecgLocal* partagée par la primitive *maj_iecg_local* avec l'agent de transfert et l'agent de localisation. *Moy* est une variable locale qui contient la moyenne des charges locales.

***Primitive maj_iecg_local (iecg)***
Début
- P(mutexIecgLocal);
- *Moy* ← *chargeMoy*;
- *iecg*Local ← *iecg*;
- V(mutexIecgLocal);

Fin.

### d) La primitive remontée_*icl*

Dans l'algorithme ci-dessous, la procédure *RPC* est un appel à distance de la primitive (ici *maj_icg*) désignée par son deuxième paramètre. Les autres paramètres de la procédure représentent, dans cet ordre, le nom du site qui exécute la primitive et les arguments de celle-ci. *calcul_L_DELAY* calcule la nouvelle valeur de L_DELAY, c'est un simple codage de l'expression de calcul de L_DELAY donnée précédemment. Les variables *icl*, *ancienIcl* et *variationIcl* repésentent respectivement l'index de charge locale actuel, celui calculé L_DELAY unités de temps auparavant et un témoin de variation significative de la charge dans le site de numéro *numSite*. La variation de charge est significative si elle est supérieure à DELTA, une valeur à fixer [7].

[7] Dans [MDLT96d], nous l'avons fixée à deux threads actifs.

L'appel à la primitive *calcul_icl* d'estimation de la charge locale est un simple appel de fonction. Il serait plus coûteux de déléguer un processus pour exécuter cette primitive.

***Primitive remontée_icl()***
booléen variationIcl ← FAUX;
entier *icl*,ancienIcl ← 0;
Début
•L_DELAY ← INIT_DELAY [8];
• Tant que (VRAI) Début
• delai(L_DELAY); /* blocage pendant L_DELAY */
• *icl* ← calcul_*icl*();
• variationIcl ← $|icl - ancienIcl| > DELTA$;
• Si (Non variationIcl) alors
•L_DELAY ← $(1+r_1)\times$L_DELAY;
sinon Début
• RPC(maître,maj_icg,numSite,*icl*);
• calcul_L_DELAY(ancienIcl,*icl*);
Fin sinon
• ancienIcl ← *icl*;
Fin Tant que.
Fin.

**e) La primitive calcul_*icl***

***Primitive calcul_icl ()***
Début
• retourner le nombre de processus actifs;
Fin.

Certains environnements de programmation intègrent cette primitive, il suffit donc de l'appeler au besoin. Par exemple, PM$^2$ met à la disposition du programmeur une primitive appelée "pthread_activethreads_np()" qui permet de calculer le nombre de threads actifs de l'application dans le site.

### 7.3.2 Agent de transfert

L'agent de transfert décide sur chaque site (décision locale) si la création d'une entité de régulation doit se faire localement ou sur un site distant. Le transfert est à l'initiative du site créateur de l'entité. De ce fait, la politique de régulation est *active*. Aucune décision de transfert ne passe par le maître. Par conséquent, *l'agent de transfert est totalement distribué*. Le critère de transfert utilise les informations de charge de la machine i.e. l'information de charge locale *icl* et l'information de charge globale *iecgLocal* produite par l'agent d'information. Un transfert est décidé sur un site si celui-ci est lourdement chargé et s'il y a au moins un site légèrement chargé dans la table *iecgLocal*.

[8] INIT_DELAY=L_DELAY($t_0$) donné dans la formule (7.6).

Dans le cas où l'index *iecgLocal* est vide, deux solutions sont possibles : soit l'entité de transfert est créée localement ou alors la création de celle-ci est différée jusqu'à ce qu'il y ait de nouveau des sites légèrement chargés. Dans [MDLT96d], les threads continuent leur exploration (recherche de solution dans un sous-arbre) séquentiellement. La création différée ne se traduit par aucun blocage de threads. Dans notre implantation de $P^3$, la solution qui consiste à créer localement la copie n'est pas intéressante. Ceci parce que pour pouvoir exploiter le parallélisme qui lui est inhérent il faudrait la migrer, ce qui est coûteux. La deuxième solution est retenue.

La création différée dans notre implantation se traduit par le blocage du thread qui effectue la copie jusqu'à ce que l'index *iecgLocal* ne soit plus vide. Deux solutions sont possibles pour mettre en œuvre ce blocage. La première solution consiste à bloquer le thread de copie pendant un certain délai fixe. Il sera alors réveillé par l'ordonnanceur MARCEL au terme de ce délai. La deuxième solution est de bloquer le thread sur un sémaphore partagé avec le thread *S_PROCESS*. Le thread de copie sera alors réveillé par *S_PROCESS* dès celui-ci reçoit un index *iecg* non vide.

### 7.3.3 Agent de localisation

#### 7.3.3.1 Description

L'agent de localisation permet de sélectionner les sites cibles des transferts décidés par l'agent de transfert. Son fonctionnement est le suivant. Dans chaque site esclave, il dispose d'une table locale *iecgLocal* de sites légèrement chargés et d'un information *next* qui indique l'entrée de *iecgLocal* qui contient le prochain site destinataire. Au lancement d'une application, la table *iecgLocal* est initialisée à tous les sites de la machine (représentés par leurs numéros). L'information *next* est initialisé à *(numSite* **modulo** *tailleIecgLocal)* où *numSite* et *tailleIecgLocal* désignent respectivement le numéro du site courant et le nombre d'entrées de la table *iecgLocal*.

A chaque transfert, le site désigné par l'information *next* est choisi comme cible. Ensuite, le site sélectionné est supprimé de la table *iecgLocal* en incrémentant l'information *next* et en décrémentant la longueur de la table. Ceci se traduit par la séquence d'instructions suivante.

- $next \leftarrow (next + 1)$ *modulo* $tailleIecgLocal$;
- $tailleIecgLocal \leftarrow tailleIecgLocal - 1$;

Le symbole "$\leftarrow$" désigne l'affectation. Ces opérations sont illustrées par la figure 7.3. Dans cette figure, *iecgLocal* contient 6 sites légèrement chargés. Il s'agit des sites 3, 4, 6, 8, 12 et 14. Les sites hachurés (6 et 8) ont déjà été désignés. Ils ne sont plus éligibles dans le site contenant la table *iecgLocal* avant la fin du délai *L_DELAY* en cours. Les autres sites (non hachurés : 3, 4, 12 et 14) sont candidats aux prochains transferts. Le site pointé par la variable *next* i.e. 12 sera exactement le prochain site à élire. Après avoir été élu, le site 12 est supprimé de la table *iecgLocal* (il est hachuré après l'opération de transfert). L'information *next* est incrémenté, la prochaine cible du transfert est maintenant le site 14.

#### 7.3.3.2 Caractéristiques

La politique de localisation de notre approche de régulation dynamique de charge a les propriétés suivantes :

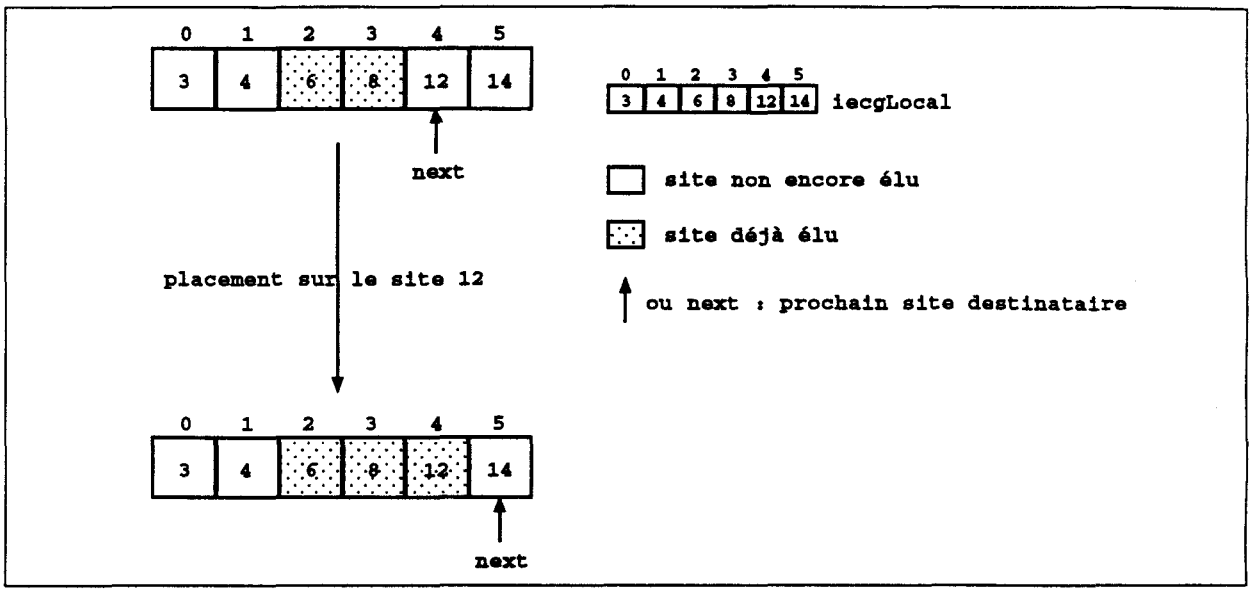



Figure 7.3 : La table *iecgLocal* d'un site avant et après un transfert

- Elle est *globale* : Dans le choix du site destinataire du transfert, on ne se limite pas au voisinage du site émetteur. Tous les sites de la machine sont considérés.
- Elle est *distribuée* : Le site destinataire est élu de façon totalement décentralisée par le site actif. C'est une politique *active*.
- Elle préserve la stabilité du système. En effet, elle minimise les risques d'inondation du site destinataire pour les raisons suivantes :
  - ▷ L'information *next* n'est pas initialisé à la même valeur pour tous les sites.
  - ▷ Le site destinataire du transfert est sélectionné une seule fois par le même site pendant une période égale à L_DELAY. Par conséquent, il reçoit au pire (n-1) entités de régulation, n étant le nombre de sites de la machine.

## 7.4 Implantation à base de threads

### 7.4.1 Implantation

Nous rappelons que la machine sous-jacente à l'implantation de notre modèle est une ferme de 16 processeurs DEC/ALPHA interconnectés par un réseau Ethernet. Nous avons adopté dans un premier temps une approche multi-tâches en utilisant l'environnement PVM [GBea94]. Le problème de cette approche est, entre autres, l'utilisation des signaux pour la gestion des délais *L_DELAY* et *G_DELAY* utilisés dans la collecte des informations de charge. En effet, PVM est non réentrant et ne se protège pas contre les signaux. Par conséquent, une approche basée sur les threads est nécessaire. C'est pourquoi, nous avons utilisé l'environnement multithreadé PM² [NM95, Nam97]. Dans l'implantation de notre approche de régulation de charge, nous avons utilisé des LRPC asynchrones : *S_PROCESS*, *M_PROCESS*, le service de mise à jour de *icg* et

le service de mise à jour de l'information globale (lors de sa diffusion) dans chaque site sont des LRPC asynchrones.

### 7.4.2 Calcul de LOW_DELAY et INIT_DELAY

Avant de donner les formules de calcul de **LOW_DELAY** et *INIT_DELAY*, définissons d'abord le concept de *session d'information.*

**Définition 3 :** Une **session d'information** est la période de temps allant de l'instant où tous les processus *S_PROCESS* remontent [9] leurs *icl* au moment où ils reçoivent l'information globale diffusée par *M_PROCESS.*

La formule donnant la durée $d_{si}$ d'une session d'information (voir figure 7.4) est la suivante :

$$d_{si} = t_r + n.t_{maj1} + t_{iecg} + t_{diff} + t_{maj2} \tag{7.5}$$

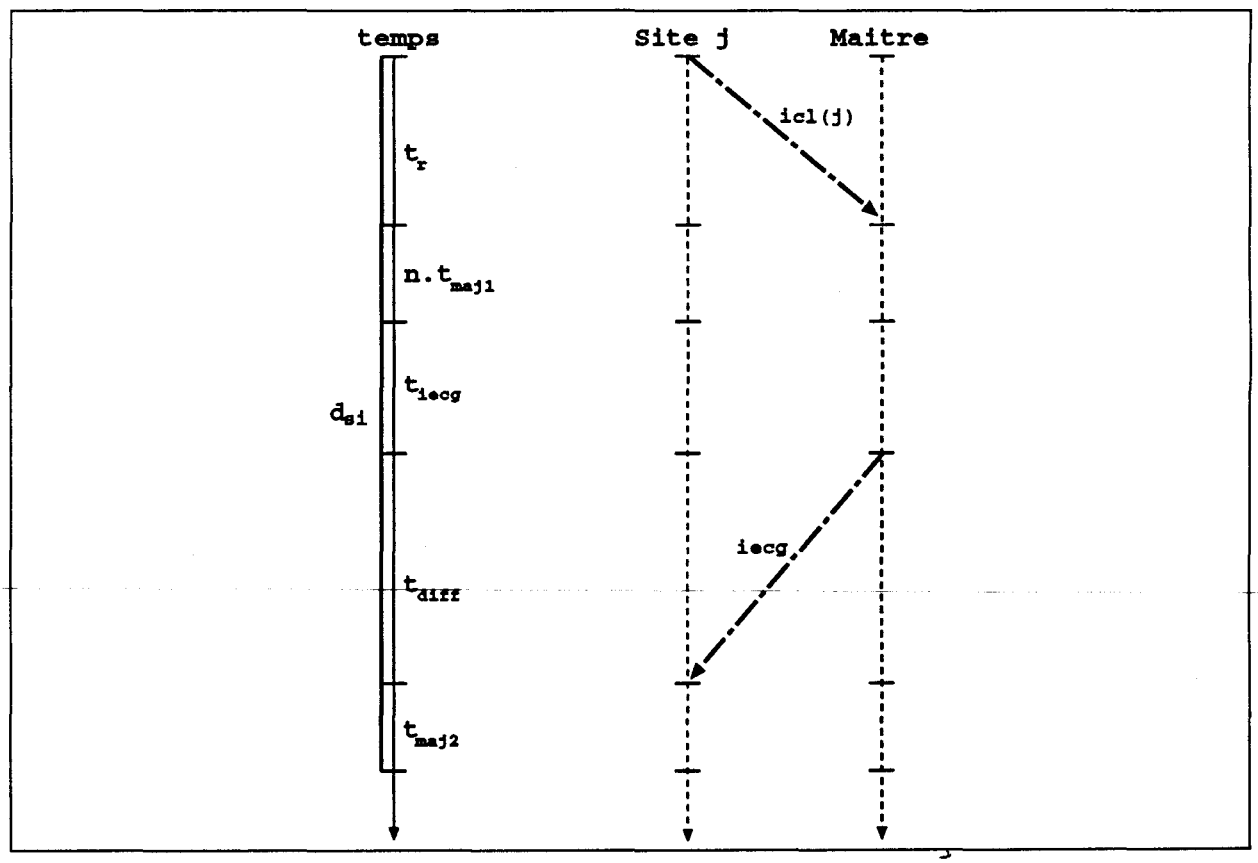



Figure 7.4 : Illustration de $d_{si}$

[9] On suppose que toutes les remontées sont faites simultanément

Les paramètres $t_r, t_{maj1}, t_{iecg}, t_{diff}, t_{maj2}$ désignent respectivement les temps de remontée par *S_PROCESS* de *icl*, de mise à jour de l'index de charge icg dans le site où se trouve *M_PROCESS*, de calcul de la charge moyenne et du vecteur *iecg* par *M_PROCESS*, de diffusion de cet index *iecg* et de réactualisation de l'*iecg* présent dans un site. Ces temps sont, d'une part, dépendants des caractéristiques de l'architecture cible de l'exécution notamment la vitesse de calcul des sites et les performances du réseau en termes de débit et de latence. D'autre part, ils sont fonction de l'environnement de programmation et de l'interface de communication utilisés.

Avec les considérations d'implantation du paragraphe précédent, nous avons mesuré $d_{si}$ en fonction du nombre $n$ de processeurs de la machine. La courbe obtenue est celle donnée par la figure 7.5. Par interpolation linéaire, cette courbe se traduit par l'équation suivante :

$$d_{si} = 2.7.n + 0.3$$

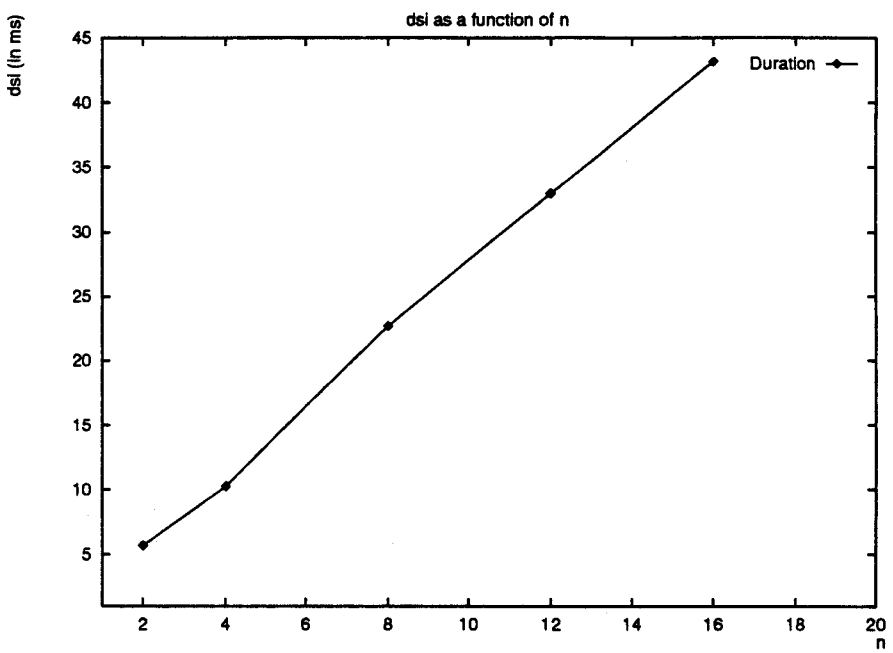



Figure 7.5 : Mesure de $d_{si}$

Les valeurs de *LOW_DELAY* et *INIT_DELAY* sont calculées en fonction de $d_{si}$ de la façon suivante :

$$LOW\_DELAY = \alpha \times d_{si} \text{ et } INIT\_DELAY = \beta \times d_{si}. \quad (7.6)$$

Dans [MDLT96d], nous avons fixé $\alpha$ et $\beta$ respectivement égaux à *2* et *10*.

## 7.5 Conclusion

Dans ce chapitre, nous avons présenté une approche de régulation dynamique de la charge. Cette approche comprend une politique adaptative et centralisée de collecte d'informations de charge. D'une part, chaque site esclave calcule sa charge locale périodiquement et ne la remonte au site maître que si celle-ci a changé de façon significative par rapport à la dernière fois où elle avait été calculée. La période (locale) de calcul est adaptative car elle est fonction de la variation de la charge dans la machine.
D'autre part, le site maître consulte sa charge globale (vecteur de charges locales) périodiquement. La période (globale) de consultation est également adaptative. Si la charge locale d'au moins un site a changé en comparaison de la dernière consultation de la charge globale alors le site maître calcule de nouveau la charge moyenne et le vecteur des processeurs légèrement chargés puis les diffuse à tous les sites esclaves.
Par ailleurs, les valeurs initiale et minimale des périodes locale et globale sont calculées en fonction de l'environnement de programmation et de l'architecture cible de l'exécution.

L'approche proposée intègre également un outil qui minimise le risque d'inondation des sites. En effet, chaque fois que l'agent de localisation sélectionne le site cible d'un transfert à partir d'un index local de sites légèrement chargés il supprime celui-ci de l'index. De plus, les sites ne commencent pas tous la sélection par le même numéro de site.

L'approche a été expérimentée sur une ferme de processeurs DEC/ALPHA en utilisant l'environnement de programmation multithreadé $PM^2$. L'évaluation théorique et expérimentale de l'approche est donnée dans le chapitre suivant.

# Chapitre 8

# Evaluation de l'algorithme

## 8.1 Introduction

Les performances de l'algorithme de régulation de charge dépendent de la qualité de l'information de charge produite par l'agent d'information et du surcoût de calcul et de communication que celui-ci induit. Elles dépendent également de la qualité des décisions prises, lors de l'exécution, par les agents de transfert et de localisation ainsi que des surcoûts que ceux-ci occasionnent. Deux types d'évaluation de performances sont possibles : une évaluation par une méthode analytique et une évaluation expérimentale. L'étude analytique permet d'exprimer les coûts des différents agents. D'autre part, l'évaluation expérimentale permet de donner une idée sur, non seulement les surcoûts induits, mais aussi sur la qualité des différents agents.

Le chapitre est structuré de la manière suivante. Avant tout, nous évaluons analytiquement notre algorithme de régulation de charge. L'étude analytique est, dans un premier temps, restreinte à l'agent d'information en calculant le surcoût de calcul et de communication qu'il induit. Ensuite, nous procédons à une évaluation expérimentale de l'algorithme sur un autre type d'application irrégulière. Il s'agit de l'application de l'algorithme IDA* au problème du taquin 15. Des mesures de performances sont également présentées. Enfin, nous terminons le chapitre par une conclusion.

## 8.2 Evaluation par étude analytique

Dans l'évaluation par étude analytique de notre algorithme de régulation dynamique et adaptative de la charge, nous n'avons considéré, pour le moment, que l'agent d'information de l'algorithme.

Dans le chapitre précédent, nous avons indiqué que dans le choix de la fréquence de collecte de l'information de charge, un compromis entre la fraîcheur de l'information et le surcoût de calcul et de communication induit par sa collecte doit être trouvé. Ce surcoût influe considérablement sur l'efficacité de l'exécution. Par conséquent, il doit être le plus faible possible. Dans ce paragraphe, nous allons déterminer la borne majoratrice du surcoût de calcul et de communication induit par l'agent d'information i.e. le surcoût dans le cas où le système est fluctuel (instable) durant

toute la durée d'exécution d'une application.

### 8.2.1 Surcoût de calcul

Etant donné que l'exécution est effectuée par les sites esclaves, nous allons déterminer le surcoût de calcul induit par l'agent d'information sur chaque site esclave. Appelons $C_s$ ce surcoût de calcul. La formule de calcul de $C_s$ est la suivante :

$$C_s = N \cdot c$$

où N est le nombre total de fois où la charge locale *icl* du site est calculée. En d'autres termes, N représente le nombre de périodes $L\_DELAY$ dans T, T étant le temps global d'exécution de l'application. Le paramètre c est la somme du coût de calcul de la charge locale et du coût de réception du vecteur *glsi*. Son calcul dépend de l'environnement d'implantation. Nous allons maintenant calculer la valeur de N.

Dans le cas où le système est fluctuel, la période $L\_DELAY$ décroit, comme le montre la figure 8.1, avec une pente égale à $r_2$ jusqu'à ce qu'elle atteigne la valeur $LOW\_DELAY$. Appelons $T_0$ l'instant où cette valeur est atteinte (voir figure 8.1). A partir de $T_0$, la période garde la valeur $LOW\_DELAY$ jusqu'à la fin de l'exécution. On peut donc distinguer deux phases : la phase avant $T_0$ et celle après $T_0$. Appelons $N_1$ (resp. $N_2$) le nombre total de périodes $L\_DELAY$ dans $T_0$ (resp. $T - T_0$). On a alors :

$$N = N_1 + N_2$$

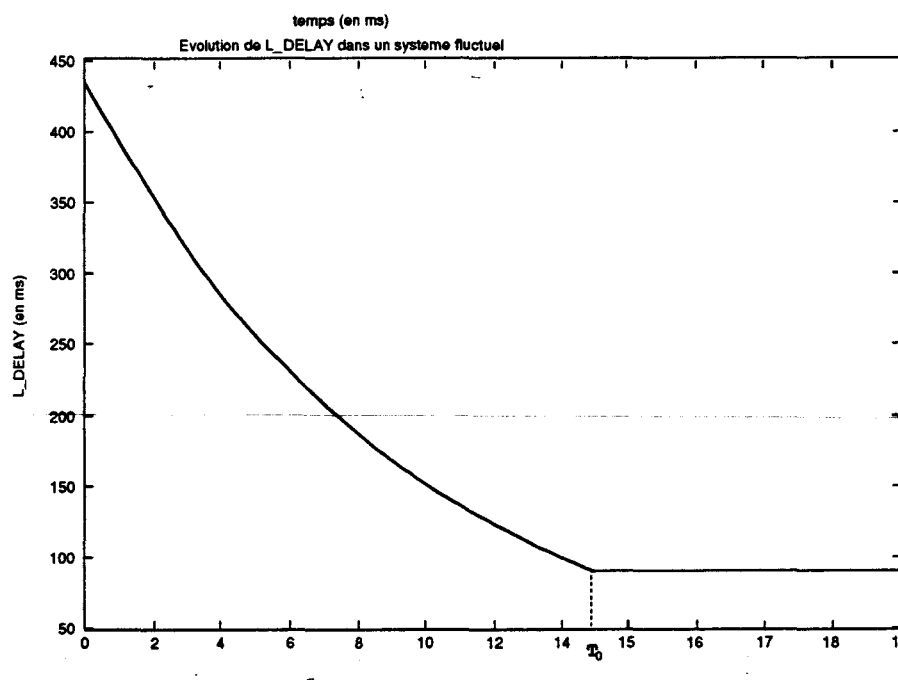



Figure 8.1 : Evolution de $L\_DELAY$ dans un système fluctuel de 16 sites

### a) Calcul de $N_1$

Soit $L\_DELAY(t_i)$ la valeur de la période $L\_DELAY$ à l'instant $t_i$ donné. En utilisant la formule de calcul de $L\_DELAY$ présentée au chapitre précédent, la valeur de $L\_DELAY$ à l'instant $t_{i+1}$ (adjacent à $t_i$) est la suivante :

$$L\_DELAY(t_{i+1}) = (1 - r_2).L\_DELAY(t_i)$$

Considérons maintenant les instants successifs $t_0$ (instant initial), $t_1$, ..., $t_{N_1} = T_0$ tels que:

$$t_{i+1} = t_i + L\_DELAY(t_i)$$

On peut écrire :

$$\begin{aligned} L\_DELAY(t_0) &= INIT\_DELAY \\ L\_DELAY(t_1) &= (1 - r_2).L\_DELAY(t_0) \\ L\_DELAY(t_2) &= (1 - r_2)^2.L\_DELAY(t_0) \\ &\cdots \\ L\_DELAY(t_{N_1}) &= (1 - r_2)^{N_1}.L\_DELAY(t_0) \end{aligned}$$

Au bout de $N_1$ périodes $L\_DELAY$, on a :

$$L\_DELAY(T_0) = (1 - r_2)^{N_1}.INIT\_DELAY \tag{8.1}$$

D'autre part, on sait que la période à l'instant $T_0$ est égale à $LOW\_DELAY$. On a donc :

$$L\_DELAY(T_0) = LOW\_DELAY \tag{8.2}$$

Par combinaison des équations 8.1 et 8.2 :

$$(1 - r_2)^{N_1}.INIT\_DELAY = LOW\_DELAY$$

Ce qui implique :

$$(1 - r_2)^{N_1} = \frac{LOW\_DELAY}{INIT\_DELAY}$$

$$N_1 = \frac{\ln(\frac{LOW\_DELAY}{INIT\_DELAY})}{\ln(1-r_2)}$$

En remplaçant les paramètres $INIT\_DELAY$ et $LOW\_DELAY$ par leurs valeurs calculées dans le chapitre précédent, on obtient :

$$N_1 = \frac{\ln(\frac{\alpha.d_{si}}{\beta.d_{si}})}{\ln(1-r_2)}$$

La valeur de $N_1$ est donnée par la formule suivante :

$$N_1 = \frac{\ln(\frac{\alpha}{\beta})}{\ln(1 - r_2)} \tag{8.3}$$

### b) Calcul de $N_2$

Etant donné que la période reste constante et égale à $LOW\_DELAY$ après $T_0$, la valeur de $N_2$ peut être exprimée par :

$$N_2 = \frac{T - T_0}{LOW\_DELAY}$$

La valeur de $T_0$ est la somme de toutes les valeurs prises par $L\_DELAY$ entre l'instant 0 et le moment où celle-ci passe à $LOW\_DELAY$ i.e. $T_0$. On rappelle qu'il y a $N_1$ valeurs. On peut donc écrire :

$$T_0 = L\_DELAY(t_0) + L\_DELAY(t_1) + ... + L\_DELAY(t_{N_1})$$

$$\Rightarrow T_0 = INIT\_DELAY + (1 - r_2).INIT\_DELAY + ... + (1 - r_2)^{N_1}.INIT\_DELAY$$

$$\Rightarrow T_0 = [1 + (1 - r_2) + ... + (1 - r_2)^{N_1}].INIT\_DELAY$$

$$\Rightarrow T_0 = \frac{1 - (1-r_2)^{1+N_1}}{1 - (1-r_2)}.INIT\_DELAY$$

En remplaçant $INIT\_DELAY$ et $N_1$ par leurs valeurs respectives, on obtient :

$$T_0 = \frac{1 - (1-r_2)^{1+\frac{\ln(\frac{\alpha}{\beta})}{\ln(1-r_2)}}}{r_2}.\beta.(2.7n + 0.3)$$

On rappelle que n est le nombre de sites dans la machine. En remplaçant $T_0$ par sa valeur dans l'expression de $N_2$, on obtient :

$$N_2 = \frac{T - (\frac{1 - (1-r_2)^{1+\frac{\ln(\frac{\alpha}{\beta})}{\ln(1-r_2)}}}{r_2}.\beta.(2.7n + 0.3))}{\alpha.(2.7n + 0.3)} \tag{8.4}$$

### c) Calcul de $C_s$

En résumé, on a :

$$C_s = N \,.\, c = (N_1 + N_2) \,.\, c$$

En remplaçant $N_1$ et $N_2$ par leurs valeurs données respectivement par les formules 8.3 et 8.4, on obtient :

$$C_s = (\frac{\ln(\frac{\alpha}{\beta})}{\ln(1 - r_2)} + \frac{T - (\frac{1 - (1-r_2)^{1+\frac{\ln(\frac{\alpha}{\beta})}{\ln(1-r_2)}}}{r_2}.\beta.(2.7n + 0.3))}{\alpha.(2.7n + 0.3)}) \,.\, c \tag{8.5}$$

On peut maintenant exprimer le pourcentage majorant p que représente $C_s$ par rapport au temps global d'exécution T. L'expression de p est la suivante :

$$p = \frac{C_s}{T} = \frac{(\frac{\ln(\frac{\alpha}{\beta})}{\ln(1-r_2)} + \frac{T - (\frac{1 - (1-r_2)^{1+\frac{\ln(\frac{\alpha}{\beta})}{\ln(1-r_2)}}}{r_2}.\beta.(2.7n+0.3))}{\alpha.(2.7n+0.3)}) \,.\, c}{T} \tag{8.6}$$

### 8.2.2 Surcoût de communication

Nous exprimons le surcoût de communication comme étant le nombre total de communications induites par la collecte d'informations de charge pour tous les sites esclaves. Pendant une période $L\_DELAY$, chaque site nécessite au maximum deux communications : une pour remonter la charge locale au site maître et une faite par le site maître pour l'envoi de *glsi*. Le nombre total $M_s$ de communications faites par chaque site est donné par la formule suivante :

$$M_s = 2.N = 2.(\frac{\ln(\frac{\alpha}{\beta})}{\ln(1-r_2)} + \frac{T - (\frac{1-(1-r_2)^{1+\frac{\ln(\frac{\alpha}{\beta})}{\ln(1-r_2)}}}{r_2}.\beta.(2.7n+0.3))}{\alpha.(2.7n+0.3)})$$

Le nombre total M de communications induites par la collecte d'informations de charge pour tous les n sites esclaves est par conséquent :

$$M = M_s.n = 2n.(\frac{\ln(\frac{\alpha}{\beta})}{\ln(1-r_2)} + \frac{T - (\frac{1-(1-r_2)^{1+\frac{\ln(\frac{\alpha}{\beta})}{\ln(1-r_2)}}}{r_2}.\beta.(2.7n+0.3))}{\alpha.(2.7n+0.3)}) \quad (8.7)$$

## 8.3 Evaluation expérimentale

La première évaluation expérimentale de notre algorithme de régulation dynamique et adaptative de la charge a été faite sur une application irrégulière autre que les langages fonctionnels. Il s'agit de l'application parallèle d'une heuristique de recherche dans un arbre à un problème combinatoire. L'heuristique et le problème sont décrits ci-dessous.

### 8.3.1 L'heuristique IDA*

IDA* ("**I**terative **D**eepening **A***") [Kor85] est un algorithme de recherche qui combine deux heuristiques bien connues en intelligence artificielle. Il s'agit de l'heuristique de recherche itérative en profondeur d'abord ("Depth-first Iterative Deepening Search") et de l'heuristique A* [Nil80].

D'une part, IDA* utilise le principe de recherche itérative de la première heuristique. En effet, celle-ci procède par itérations i.e. elle effectue une recherche jusqu'à la profondeur 1, puis la profondeur 2 et ainsi de suite jusqu'à ce que la solution recherchée soit trouvée. La profondeur de l'arbre représente le seuil limite de chaque itération. L'algorithme IDA* utilise le même principe mais au lieu de limiter la profondeur de la recherche, on limite plutôt le coût de celle-ci. Ainsi, un nœud de l'arbre de recherche est explorable si son coût associé est inférieur ou égal au seuil de l'itération courante. Le seuil est calculé de la façon suivante : le seuil de la première itération est égal au coût associé à la racine de l'arbre. Le seuil d'une quelconque autre itération est le minimum des coûts ayant dépassé le seuil de l'itération précédente.

D'autre part, IDA* utilise la même fonction de calcul de coût que celle utilisée par l'heuristique A*. Cette fonction est définie par : le coût associé à un nœud donné n de l'arbre est $f(n) = g(n) + h(n)$. Dans cette formule, $g(n)$ est le coût du chemin parcouru de la racine de l'arbre jusqu'à n et $h(n)$ est le coût minorant estimé du chemin à parcourir de n jusqu'à la solution

du problème. La valeur $g(n)$ représente la profondeur du nœud n dans l'arbre. La valeur $h(n)$ dépend du problème traité.

Dans IDA*, le parallélisme peut être considéré en explorant en parallèle différents sous-arbres. Il a été montré que IDA* pallie la contrainte d'espace mémoire de l'algorithme A* et permet de trouver la solution optimale (de moindre coût).

### 8.3.2 Le problème du taquin 15

Le taquin 15 est une grille carrée 4x4. Quinze parmi les cases de la grille contiennent chacune un nombre compris entre 1 et 15. La seizième case est vide et est applelée *blanc.*

Le problème du taquin 15 consiste, à transformer une configuration initiale de la grille, par des déplacements successifs du blanc vers les cases adjacentes, en une configuration finale. Dans la configuration initiale, les nombres sont placés dans les cases dans un ordre quelconque. La configuration du taquin est telle que les nombres sont palcés dans l'ordre croissant. Par exemple, la figure 8.2 illustre une instance du problème du taquin 15. Elle représente le $12^{eme}$ des 100 problèmes générés aléatoirement par Korf [Kor85].

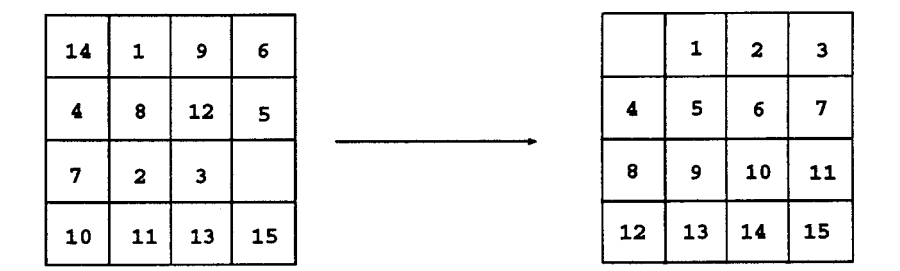



Figure 8.2 : La $12^{eme}$ instance du problème du taquin 15

### 8.3.3 Application : IDA* appliqué au problème du taquin 15

Lors de la transformation de la configuration initiale du problème du taquin 15, chaque mouvement du blanc conduit à de nouvelles configurations. Par exemple, dans la figure 8.3.(a), les trois configurations, de gauche à droite, sont obtenues par mouvement du blanc dans cet ordre vers le haut, à gauche et vers le bas. Par ailleurs, les différentes configurations générées peuvent être représentées, comme le montre la figure 8.3.(b), par un arbre de recherche où chaque nœud désigne une configuration. Le nœud associé à la configuration finale représente la solution du problème du taquin 15.

L'arbre de recherche associé au problème du taquin 15 peut être exploré par l'algorithme IDA*. Dans la fonction coût de l'algorithme, la valeur $g(n)$ représente la profondeur de n i.e. le nombre de mouvements du blanc effectués pour passer de la configuration initiale à la configuration associée au nœud n. D'autre part, la valeur $h(n)$ représente la somme des distances

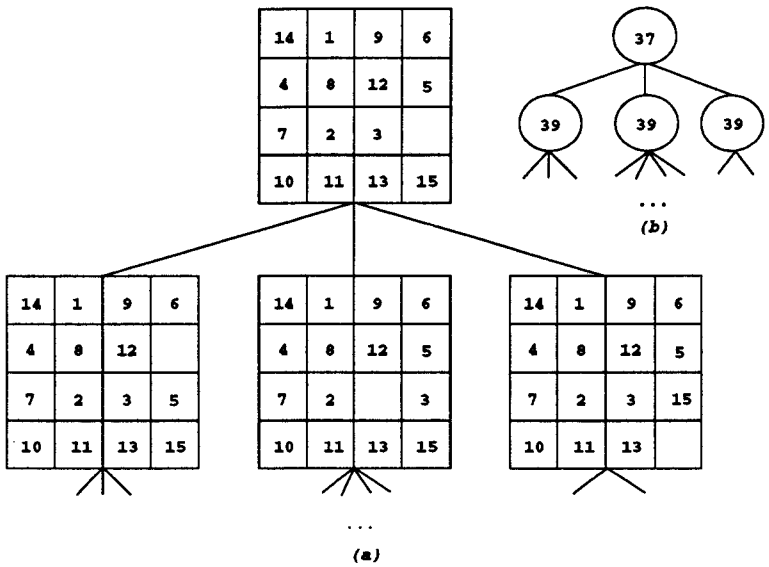



Figure 8.3 : Représentation du problème du taquin 15 par un arbre de recherche

de Manhattan [Nil80] de toutes les cases (le blanc non compris) de la grille associée au nœud n. La distance de Manhattan d'une case c contenenant un nombre donné i ($i \in [1,15]$) est la somme $l_1 + c_1$. Dans cette somme, $l_1$ (resp. $c_1$) désigne la différence absolue entre le numéro de ligne (resp. de colonne) de la case c et celui de la case déstination (finale) du nombre i.

### 8.3.4 Expérimentation

#### 8.3.4.1 Implémentation

Nous avons implémenté une version multithreadée de l'application ci-dessus sur une ferme de 16 processeurs DEC/ALPHA en utilisant le modèle de programmation PM$^2$. L'exploration de l'espace de recherche du problème du taquin 15 est assurée par des LRPC avec attente différée. Initialement, un seul thread est créé pour lancer l'exploration à la racine de l'arbre de recherche. Ensuite, ce thread crée d'autres threads pour explorer différents sous-arbres. Ces threads génèrent, à leur tour, d'autres threads et ainsi de suite jusqu'à ce que la solution du problème soit trouvée. Deux grands problèmes doivent être pris en charge concernant ces différents threads :

- Quelle est la taille du sous-arbre exploré par un thread?
- Comment sont distribués les threads sur les différents sites de la machine?

Ces deux questions désignent respectivement les problèmes de granularité de parallélisme et de régulation dynamique de la charge. Ils sont traités ci-dessous.

### 8.3.4.2 Granularité de parallélisme

Pour gérer le problème de granularité de parallélisme, nous avons identifié deux types de threads d'exploration : les threads *fertiles* et les threads *stériles*. Par opposition aux threads fertiles, les threads stériles explorent chacun un sous-arbre de façon séquentielle sans générer d'autres threads i.e. sans appeler l'élément de transfert du régulateur de charge. Le critère d'identification d'un thread stérile est le suivant : la racine du sous-arbre exploré par un thread stérile vérifie la condition suivante :

$$|g(n) + h(n) - Seuil\ de\ l'iteration| \leq G$$

Cette condition détermine la granularité des threads stériles i.e. la taille maximale du sous-arbre exploré par chaque thread stérile. Elle peut être vue comme une condition d'arrêt de la génération de threads. Par ailleurs, le paramètre G est déterminé par une série de 100 exécutions de l'algorithme IDA* au problème de Korf numéro 12, illustré ci-dessus. La figure 8.4 montre l'influence du paramètre G sur le speed-up relatif de l'exécution. Ce dernier est le rapport entre le temps d'exécution parallèle sur un site et le temps d'exécution sur 16 sites.

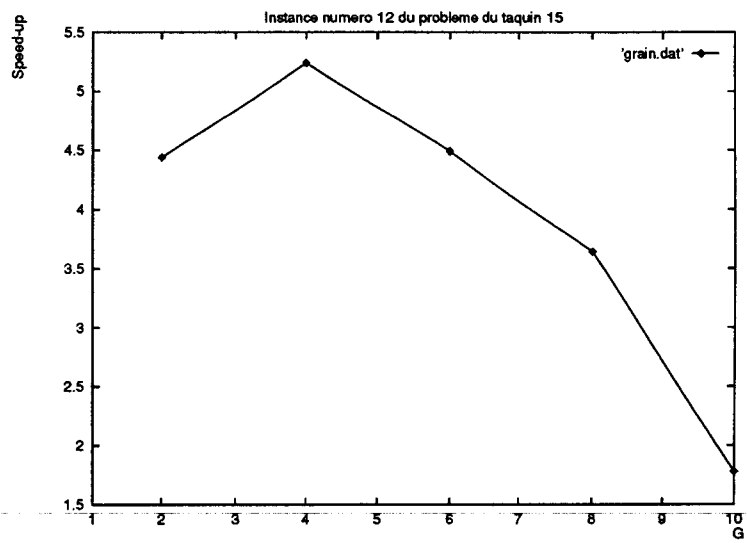



Figure 8.4 : Influence de la taille du grain d'un thread sur le speed-up

La valeur $G = 4$ donne le meilleur speed-up. D'une part, pour les valeurs de G inférieures à 4, le speed-up n'est pas important car la granularité des threads stériles est tellement petite qu'elle induit un surcoût de communication non négligeable. D'autre part, pour les valeurs de G supérieures à 4, le speed-up décroit avec la croissance de G car le parallélisme est de moins en moins exploité. Nous avons retenu la valeur $G = 4$ dans les mesures que nous allons présenter un peu plus loin.

#### 8.3.4.3 Régulation dynamique de la charge

Le problème de régulation de charge concerne tous les threads d'exploration (fertiles et stériles) générés pendant la recherche de solution au problème du taquin 15. L'algorithme de régulation de charge décrit dans le chapitre précédent est utilisé pour répartir ces threads sur les différents sites de la machine en fonction de la charge de ces derniers. La fonction de caractérisation de la charge d'un site utilisée par l'agent d'information de l'algorithme est la suivante : la charge d'un site est égale au nombre de threads d'exploration créés dans ce site. Par ailleurs, le seuil de sous-charge utilisé par l'agent de transfert est fixé empiriquement à 5 threads. Un site est déclaré légèrement chargé si sa charge locale est inférieure à 5.

### 8.3.5 Evaluation des performances

Afin d'évaluer les performances de notre algorithme de régulation de charge, nous avons expérimenté celui-ci sur quelques instances du problème du taquin 15 générées, de façon aléatoire, par Korf dans [Kor85]. Pour le choix des instances à résoudre, nous avons d'abord identifié trois classes de configurations : *Légère*, *Normale* et *Lourde*. Le critère de classification est le temps d'exploration séquentielle (sur un seul site). Ces trois classes contiennent des instances dont le temps d'exploration est respectivement de quelques secondes, quelques minutes et quelques heures. Comme le montre la table 8.1, trois configurations sont choisies de chaque classe. Celles de la classe *lourde* sont les trois plus longues.

Tableau 8.1 : IDA* appliqué au problème du taquin 15 : résultats préliminaires

| Classe | Instances du problème | speed-up relatif moyen | Efficacité |
|---|---|---|---|
| *Légère* | 28, 78, 23 | 6.45 | 0.40 |
| *Moyenne* | 27, 32, 63 | 12.45 | 0.77 |
| *Lourde* | 60, 82, 88 | 14.83 | 0.92 |

La table 8.1 montre le speed-up et l'efficacité relatifs moyens obtenus avec les instances du problème choisies. Le speed-up relatif est calculé de la même manière que dans le paragraphe 8.3.4.2. L'efficacité est égale au rapport du speed-up relatif au nombre total de sites i.e. 16. Les moyennes sont calculées sur 100 exécutions de chaque instance du problème pour chaque classe.

Les résultats montrent que le speed-up et l'efficacité relatifs de la classe "Légère" sont faibles. Ceci parce que le temps passé dans la distribution du travail est important en comparaison du temps global d'exploration. Les résultats montrent également que le speed-up et l'efficacité relatifs sont d'autant plus important que le temps global de recherche de la solution est important. Ils sont plus importants pour la classe "Lourde". Pour cette classe de configurations, les communications induites par la régulation de charge sont recouvertes par l'exploration de l'espace de recherche. Ce recouvrement est rendu plus conséquent par l'utilisation de l'approche à base de threads de l'implémentation.

## 8.4 Conclusion

Dans ce chapitre, nous avons donné une formulation mathématique du surcoût de calcul et de communication occasionné par la politique de collecte d'informations de charge. Ce surcoût dépend des paramètres de la politique, du nombre de sites de la machine, du temps global d'exécution de l'application ainsi que du coût de calcul de la charge locale d'un site et de celui de la mise à jour de l'information globale diffusée par le site maître.

Nous avons également présenté une évaluation expérimentale de l'algorithme sur une version multithreadée de l'application irrégulière "IDA* appliqué au problème du taquin 15". L'expérimentation a été faite sur une ferme de 16 processeurs DEC/ALPHA en utilisant le modèle de programmation $PM^2$. Un speed-up de 14.8 a été obtenu sur les instances du problème du taquin 15 les plus lourdes, ce qui est encourageant.

Dans le futur, nous évaluerons notre algorithme sur le reste des 100 instances de Korf du problème du taquin 15 afin de comparer ses performances avec celles obtenues, par exemple, dans [HTG95]. Nous allons aussi évaluer l'algorithme sur ce qui fait le cadre de notre thèse, à savoir l'implantation du modèle $P^3$.

# Conclusions et perspectives

Les langages fonctionnels suscitent beaucoup d'intérêt pour la programmation d'applications parallèles car ils se caractérisent par un potentiel de parallélisme important. Cependant, cet intérêt n'est pratiquement viable que si ce parallélisme potentiel est exploité de manière efficace. Dans cet esprit, le modèle $P^3$ d'évaluation parallèle des langages fonctionnels sans variable a été défini dans [Dev90]. Celui-ci a été étendu et validé par simulation dans [Ben94].

La simulation du modèle a, d'une part, révélé l'importance du parallélisme potentiel de celui-ci. D'autre part, elle a mis en évidence certains problèmes dont les principaux sont : la gestion de la granularité de parallélisme, la régulation dynamique de charge et l'approche interprétée utilisée pour l'exécution de programmes. Dans le cadre de cette thèse, nous avons traité ces trois problèmes de façon à répondre aux objectifs fixés dans l'introduction.

Nous avons proposé une approche originale de gestion de la granularité. Celle-ci est implicite car elle ne nécessite aucune contribution du programmeur. Elle consiste en une transformation statique du programme. D'une part, celle-ci permet de regrouper les fonctions en paquets. De plus, ce regroupement peut être révisé pendant l'exécution du programme en fonction de la taille des données. D'autre part, la notion de fenêtre traduit implicitement un regroupement "intelligent" des nœuds de données. Celui-ci est concrétisé à l'exécution du programme grâce au *mécanisme de copie* lors de la création dynamique des données. Un autre rôle important de la notion de fenêtre est l'optimisation des copies. En effet, seules les fenêtres de données (et éventuellement les données attachées à leurs feuilles) sont copiées. Le troisième type de regroupement visé par la thèse est l'association des paquets avec leurs fenêtres. Celle-ci est réalisée à l'exécution par des appels procéduraux grâce à la présence du code du programme sur tous les sites de la machine.

La méthode de gestion de la granularité conduit à une nouvelle approche d'implantation du modèle $P^3$ sur des architectures MIMD à mémoire distribuée. Le modèle de programmation supportant l'implantation est un modèle basé sur le multithreading distribué. Il utilise deux librairies : la première contient toutes les fonctions du langage Graal implémentées selon $P^3$ ; la deuxième librairie est le résultat de la traduction du programme dans un langage intermédiaire compilable distribué et multithreadé. Elle contient les fonctions implémentant les différents paquets (ou modules) du programme. Ceci est une réponse à l'objectif d'avoir une approche compilée de l'exécution.

Par ailleurs, seul le parallélisme conservatif est exploité. D'une part, le parallélisme horizontal, identifié dans l'approche de gestion de la granularité par les formes fonctionnelles et les fonctions polyadiques, est exploité en tenant compte de la taille des fenêtres. D'autre part, le parallélisme

vertical déterminé par la présence de fonctions non strictes est exploité en fonction de l'état de charge courant de la machine.

Le mécanisme de copie utilise, d'une part, une approche guidée par la notion de fenêtre. D'autre part, il reprend la technique basée sur un seuil de regroupement/éclatement définie dans [Ben94, BM95]. Ceci permet un réajustement dynamique de la répartition des données impliquant une bonne granularité des données et un bon degré de parallélisme. La copie proprement dite de la donnée est complexe car elle dépend de la taille de celle-ci ainsi que de sa répartition courante. C'est pourquoi, elle nécessite un effort considérable de synchronisation.

La politique d'ordonnancement de threads proposée utilise des classes de priorités. Ces dernières sont attribuées aux threads suivant une graduation exponentielle. Les threads de traitement générateurs de copies ont une priorité adaptative. Celle-ci dépend, à l'instar du mécanisme d'étranglement, de l'état de charge de la machine. Elle permet un bon contrôle de la création de données en ce sens qu'elle ralentit celle-ci quand la machine est surchargée et elle l'accélère dans le cas contraire.

Dans le modèle $P^3$, les données sont synonymes de traitements. La répartition de ces derniers est, par conséquent, liée à celles des données. Comme celles-ci sont dynamiques et irrégulières (les arbres de données changent constamment en forme et en taille), leur répartition doit être dynamique. C'est pourquoi, nous avons proposé un algorithme dynamique de régulation de charge.

Notre algorithme utilise une politique adaptative et centralisée de collecte d'informations de charge. D'une part, le calcul des charges locales et la consultation de la charge globale sont périodiques. Les périodes sont adaptatives et afin d'éviter un écroulement du système, elles sont minorées. De plus, leurs valeurs initiale et minimale sont calculées en fonction de l'environnement d'implantation. D'autre part, l'échange des informations de charge est relatif. Par ailleurs, les décisions de transfert et de localisation sont totalement décentralisées. Les transferts sont décidés sur la base de l'état de charge local et d'un index global de sites légèrement chargés. Cet index est utilisé localement par la politique de localisation pour choisir les sites cibles des transferts. Afin de minimiser les risques d'inondation de sites, à chaque transfert, chaque site sélectionné est retiré de l'index. De plus, les sites ne commencent pas tous la sélection par le même numéro de site.

Pour évaluer l'algorithme, nous avons, d'une part, donné une formulation mathématique des bornes majoratrices de la proportion que représente le surcoût de calcul par rapport à la durée globale d'exécution et du nombre total de communications occasionnées par la politique d'information. Ce surcoût dépend des paramètres de cette politique, du nombre de sites de la machine, du temps global d'exécution de l'application ainsi que du coût de calcul de la charge locale d'un site et de celui de la mise à jour au niveau local de l'information globale diffusée. D'autre part, nous avons évalué l'algorithme sur une version multithreadée de l'application irrégulière "IDA* appliqué au problème du taquin 15". L'expérimentation a été faite sur une ferme de 16 processeurs DEC/ALPHA en utilisant $PM^2$. Un speed-up de 14.8 a été obtenu sur les instances du problème les plus lourdes, ce qui est encourageant.

# Perspectives

Les investigations à court terme que nous envisageons dans le futur sont résumées ci-dessous :

## Gestion de la granularité

Evaluation, sur quelques programmes, de notre approche de gestion de la granularité en considérant des algorithmes aveugles (aléatoires et cycliques) de régulation de charge. Ceci nécessite, dans un premier temps, de finir l'implantation du modèle, qui est en cours.

## Régulation de charge

- Evaluation expérimentale de l'algorithme proposé dans le contexte de l'implantation du modèle $P^3$ par :
  - Comparaison avec des algorithmes aveugles ;
  - Comparaison avec d'autres algorithmes "intelligents" tels que [XH91].
- Extension de l'algorithme au modèle hiérarchique pour améliorer son extensibilité. L'agent d'information d'un tel algorithme pourrait être obtenu par utilisation récursive de l'agent d'information proposé dans chaque couche de l'algorithme hiérarchique.

Une autre extension que nous envisageons à court terme est l'intégration d'un ramasse-miettes dans l'implantation. Les algorithmes à compteurs de références proposés dans [Bev87, WW87] et [Les89] sont connus pour être de bons candidats dans le cadre de la réduction parallèle de graphe. Leur utilisation dans notre implantation est tout à fait envisageable.

Les perspectives de notre travail sont les suivantes :

- L'idée axiale du projet PARALF est l'évaluation parallèle des langages fonctionnels basée sur la représentation séparée du programme et de la donnée. Nous rappelons qu'une validation théorique de cette idée a été faite dans [Ben94] mais celle-ci reste insuffisante. En effet, une validation expérimentale est nécessaire. Pour ce faire, nous prévoyons de comparer expérimentalement le modèle $P^3$ au modèle de réduction de graphe, celui-ci étant à la base d'une panoplie d'implémentations parallèles des langages fonctionnels. L'objectif animant cette comparaison est de montrer que notre modèle peut être une alternative compétitive aux modèles de réduction utilisant une représentation uni-spaciale du programme et de sa donnée. Par ailleurs, les idées de l'approche de gestion de la granularité proposée pourraient être utilisées par d'autres implantations basées sur le modèle $P^3$.
- L'algorithme de régulation de charge a été conçu pour être général. Nous avons montré la possibilité de son utilisation pour un autre type d'application, il pourrait être réutilisé dans d'autres applications. Par ailleurs, sa politique adaptative de collecte d'informations de charge pourrait être intégrée dans d'autres algorithmes de régulation de charge. Elle pourrait être également utilisée dans d'autres applications telles que le "monitoring", le déclenchement d'un ramasse-miettes global, etc.

# Annexes

# A

# Autres fonctions du langage GRAAL

## A.1 Les fonctions "ordinaires"

### A.1.1 Les fonctions primitives

**Les fonctions de manipulation de listes**

1. La fonction **list**

   $list : a_1\ a_2\ \ldots\ a_n \quad \Longrightarrow \quad <a_1\ a_2\ \ldots\ a_n>$

2. La fonction **cons**

   $cons : \mathrm{a} < a_1\ a_2\ \ldots\ a_n > \quad \Longrightarrow \quad < a\ a_1\ a_2\ \ldots\ a_n >$

3. La fonction **append**

   $append : < x_1\ x_2\ \ldots\ x_n > < y_1\ y_2\ \ldots\ y_m > \Longrightarrow < x_1\ x_2\ \ldots\ x_n\ y_1\ y_2\ \ldots\ y_m>$

4. La fonction **length**

   $length : < a_1\ a_2\ \ldots\ a_n > \quad \Longrightarrow \quad n$
   $length : < > \quad \Longrightarrow \quad 0$

**La fonction identité** : *id*

$id : \mathrm{x} \quad \Longrightarrow \quad \mathrm{x}$

**Les fonctions prédicats**
Le langage GRAAL dispose de plus d'une centaine de fonctions prédicats. Nous en présentons quelques unes ci-dessous.

Le prédicat **null**

*null* : L $\Longrightarrow$ True si *L* est une liste vide
$\Longrightarrow$ False sinon

Le prédicat **eq**

*eq* : < x y > $\Longrightarrow$ True si x=y
$\Longrightarrow$ False sinon

Le prédicat **atomp**

*atomp* : s $\Longrightarrow$ True si *s* est un atome
$\Longrightarrow$ False sinon

Le prédicat **listp**

*listp* : s $\Longrightarrow$ True si *s* est une liste
$\Longrightarrow$ False sinon

Le prédicat **formp**

*formp* : s $\Longrightarrow$ True si *s* est un symbole de forme fonctionnelle
$\Longrightarrow$ False sinon

### A.1.2 Les formes fonctionnelles

1. La forme constante */*

   */x* : y $\Longrightarrow$ *x*

2. La forme conditionnelle **if**

   *(if p f g)* : x $\Longrightarrow$ f : x si p : x $\neq$ <>
   $\Longrightarrow$ g : x si p : x = <>

3. La forme **while**

   *(while p f)* : x $\Longrightarrow$ (if p ((while p f) $\circ$ f) id) : x

4. Les formes **binaire-unaires**
   Il existe deux formes fonctionnelles *binaire-unaires* : *binu* et *binul*. Ces deux formes ont chacune deux paramètres et s'appliquent à un seul argument. Le premier paramètre est nécessairement une fonction, le second est un objet quelconque.

   La forme **binu**

   (*binu* f x): y $\Longrightarrow$ f: x y

La forme **binul**

(*binul* f x): y $\Longrightarrow$ f: y x

5. La forme **cond**

(*cond* $f_1\ g_1\ f_2\ g_2\ \ldots\ f_n\ g_n$) : $a_1\ a_2\ \ldots\ a_n$
$\Longrightarrow$ $g_1$: $a_1\ a_2\ \ldots\ a_n$ si $f_1$: $a_1\ a_2\ \ldots\ a_n \neq$ nil
$\Longrightarrow$ $g_2$:$a_1\ a_2\ \ldots\ a_n$ si $f_2$:$a_1\ a_2\ \ldots\ a_n \neq$ nil
...
$\Longrightarrow$ $g_n$: $a_1\ a_2\ \ldots\ a_n$ si $f_n$:$a_1\ a_2\ \ldots\ a_n \neq$ nil
$\Longrightarrow$ nil

6. La forme '::'

( f :: $a_1\ a_2\ \ldots\ a_n$) : $b_1\ b_2\ \ldots\ b_k \Longrightarrow$ f : $a_1\ a_2\ \ldots\ a_n$

## A.2 Les fonctions d'ordre supérieur

Les fonctions d'ordre supérieur peuvent être des fonctions primitives, des formes fonctionnelles ou des définitions. Les fonctions primitives d'ordre supérieur sont de deux classes : les *combinateurs* et les *fonctions de méta-évaluation.* Ces deux classes sont présentées ci-dessous.

### A.2.1 Les combinateurs

Un combinateur est une fonction dont les arguments sont des fonctions quelconques de l'ensemble F et dont l'application à ses arguments fournit comme résultat une forme fonctionnelle. Les paramètres de celle-ci sont les arguments du combinateur. On peut dire que dans GRAAL, il existe autant de combinateurs que de formes fonctionnelles prédéfinies. Ci-dessous, nous donnons quelques exemples de combinateurs.

1. Le combinateur **if**
   Le combinateur *if* permet l'obtention de la forme fonctionnelle "conditionnelle" if.
   *if*: p f g $\Longrightarrow$ (if p f g)

2. Le combinateur **comp**
   Le combinateur *comp* appliqué à plusieurs fonctions permet d'obtenir la composition de ces fonctions.
   *comp* : $f_1\ f_2\ \ldots\ f_n \Longrightarrow \{\ f_1\ f_2\ \ldots\ f_n\ \}$

**Remarque** : si le nombre d'arguments du combinateur *comp* est 2 alors on aura :
*comp* : f g $\Longrightarrow$ { f g } ou f $\circ$ g

### A.2.2 Les fonctions de méta-évaluation

Les fonctions de méta-évaluation permettent de forcer l'évaluation d'une expression [1] en cours d'exécution. GRAAL dispose de trois fonctions de *métaévaluation* : *eval, apply, et funcall.* Elles sont présentées ci-dessous.

1. La fonction **eval**
   La fonction *eval* se présente sous la forme : (***eval :*** ***e***) où *e* est une expression fonctionnelle GRAAL. Le résultat de l'application est le résultat de l'évaluation de l'argument *e*. Nous avons les deux situations suivantes :

   (a) Cas où *e* est un symbole ou un nombre : le résultat de l'application de la fonction *eval* est le symbole ou le nombre lui-même.

   (b) Cas où *e* est une application de la forme : (F : $v_1$ $v_2$ ... $v_n$) où F, $v_1$ $v_2$ ... $v_n$ sont des expressions :
   $eval : e \equiv eval : (F : v_1 \ldots v_n) \equiv (eval\ F) : (eval\ v_1) \ldots (eval\ v_n)$

2. La fonction **apply**

   $$apply : \mathrm{f} < a_1\ a_2\ \ldots\ a_n > \Longrightarrow \mathrm{f} : a_1\ a_2\ \ldots\ a_n$$

   **Remarque** : l'arité de la fonction *f* doit être égale à la longueur de la liste $<a_1\ a_2\ \ldots\ a_n>$.

3. La fonction **funcall**
   Soit *f* une fonction quelconque et *a* un objet quelconque.

   $$funcall : \mathrm{f\ a} \Longrightarrow \mathrm{f} : \mathrm{a}$$

   **Remarque** : la fonction *funcall* est un cas particulier de la fonction *apply.*

   $$\mathrm{funcall} : \mathrm{f\ a} \Longleftrightarrow \mathrm{apply} : \mathrm{f} < a >$$

## A.3 Les fonctionnelles et les métaformes

Les fonctionnelles servent à combiner des fonctions pour définir de nouvelles formes fonctionnelles. Elles sont utilisées quand les formes fonctionnelles prédéfinies ne répondent pas aux besoins du programmeur. Elles se définissent par la fonction *user*, celle-ci étant représentée par le symbole **dc**.

*dc nom_fonctionnelle corps_fonctionnelle*

Lorsqu'une fonctionnelle apparaît dans un programme, la forme fonctionnelle associée est calculée dynamiquement à l'exécution. Par conséquent, dans le cas où elle apparaît dans une définition récursive sa forme fonctionnelle correspondante est recalculée plusieurs fois, ce qui est pénalisant. Dans ce cas, il est plus intéressant d'utiliser un autre type de forme : la *métaforme.* En effet, le calcul du corps de celle-ci est fait une seule fois à l'exécution. La métaforme se définit par la fonction *meta* représentée par le symbole **dm.**

*dm nom_métaforme corps_métaforme*

[1] L'expression peut être un objet simple ou une application.

# B

# Glossaire

**DFCF** : Descripteur de Fenêtre Cible d'une Fonction.

**DFCT** : Descripteur de Fenêtre Cible d'un Tronçon.

**FCF** : Fenêtre Cible d'une Fonction.

**FCP** : Fenêtre Cible d'un Paquet.

**FCT** : Fenêtre Cible d'un Tronçon.

**FRF** : Fenêtre Résultat d'une Fonction.

**FRP** : Fenêtre Résultat d'un Paquet.

**FRT** : Fenêtre Résultat d'un Tronçon.

**FTA** : File de Threads Actifs.

**FTP** : File de Threads Potentiels.

**G_DELAY** : Période de consultation de icg (délai global).

**GC** : Ramasse miettes (Garbage Collector).

**icg** : index de charge globale.

**icl** : indicateur de charge locale.

**iecg** : index d'état de charge global.

**iecgLocal** : version locale de l'index d'état de charge global.

**IDA*** : heuristique de recherche (Iterative Deepening A*).

**INIT_DELAY** : Valeur initiale des délais L_DELAY et G_DELAY.

**LFSV** : Langage Fonctionnel sans variable.

**L_DELAY** : Période de calcul de icl (délai local).

**LOW_DELAY** : Valeur minoratrice des délais L_DELAY et G_DELAY.

**LRPC** : LRPC léger (Lightweight Remote Procedure Call).

**$P^3$** : Parallel x Parallel x Parallel.

**$PM^2$** : Parallel Multithreaded Machine.

**REA** : Règle d'Exécution Abstraite.

**WHNF** : Forme normale à tête faible (Weak Head Normal Form).

# Liste des figures

# Table des tableaux

# Références


[ABF93] G. Aharoni, A. Barak, and Y. Farber. An adaptive granularity control algorithm for the parallel execution of functional programs. *FCGS*, pages 163–174, Sep 1993.

[Agh86] G. Agha. Actors. A Model of Concurrent Computation in Distributed Systems. *The MIT Press*, 1986.

[AMR92] S.E. Abdullahi, E.E. Miranda, and G.A. Ringwood. Distributed Garbage Collection. *In Int. Workshop on Memory Management, Springer-Verlag LNCS 637, St Malo, France*, Sep 1992.

[AN87] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *In PARLE'87, LNCS, Springer-Verlag*, 1987.

[AP88] F. André and J-L. Pazat. Le placement de tâches sur des architectures parallèles. *Technique et Science Informatique*, Vol.7(No.4):385–401, 1988.

[Bac78] J. W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Com. ACM*, Vol. 21(No. 8), Aug 1978.

[Ban87] S.A. Banawan. An Evaluation of Load Sharing in Locally Distributed Systems. Technical report, Aug 1987.

[BATM94] P. Bessiere, J-M. Ahuactzin, E-G. Talbi, and E. Mazer. The Ariadne's Clew Algorithm: Global Planning with Local Methods. *Proc. of the Workshop on the Algorithmic Foundations of Robotics (WAFR'94)*, pages 39–47, 1994.

[BCS89] A. Billionnet, M.C. Costa, and A. Stutter. Les problèmes de placement dans les systèmes distribués. *TSI*, 04/89/307–31, 1989.

[BDLT93] N. Bennani, N. Devesa, M.P. Lecouffe, and B. Toursel. A comparaison between the graph reduction model and an original parallel evaluation scheme for functional languages. *Euromocro Workshop on Parallel and Distributed Processing, Gran Canaria-Spain*, Jan 1993.

[Bel86] P. Bellot. *Sur les sentiers de GRAAL : étude, conception et réalisation d'un langage de programmation sans variable.* PhD thesis, Université de Paris VI, 1986.

[Ben94] N. Bennani. *Proposition d'un modèle d'évaluation parallèle des langages fonctionnels sans variables.* PhD thesis, Université de LilleI, 1994.

[Bev87] D.I. Bevan. Distributed Garbage Collection Using Reference Counting. *PARLE'87*, Springer-Verlag LNCS 259:313–321, 1987.

[BF96] Guy Bernard and Bertil Folliot. Caractéristiques Générales du Placement Dynamique : Synthèse et Problématique. *Ecole Placement Dynamique et Répartition de Charge, France*, pages 3–22, Jul 1996.

[BHLH+93] M. Beemster, P.H. Hartel, R.F.H. Hofman L.O. Hertzberger, K.G. Langendoen, L.L. Li, R. Milikowski, W.G. Vree, H.P. Barendregt, and J.C. Mulder. Experience With a Clustered Parallel Reduction Machine. *FCGS*, pages 175–200, Sep 1993.

[BM95] N. Bennani and N. Melab. Simulation d'un algorithme de répartition dynamique de données par regroupement. *Poster, Actes des 7es Rencontres Francophones du Parallélisme, RenPar'7. Mons, Belgique*, jun 1995.

[Bok81a] S.H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C–30(No.3):207–214, Mar 1981.

[Bok81b] S.H. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Transactions on Software Engeneering*, Vol.SE–7(No.6):583–589, Nov 1981.

[Bou92] P. Bouvry. Allocation de tâches dans un réseau de transputers. *Actes des 4es Rencontres Francophones du Parallélisme, RenPar'4. Lille, France*, Mar 1992.

[BS81] F.W. Burton and M.R. Sleep. Executing functional programs on a virtual tree of processors. *In FPCA'81*, pages 187–194, 1981.

[BTV92] P. Bouvry, D. Trystram, and F. Vincent. An automatic mapping tool for parallel task graphs. *Rapport technique LMC–IMAG*, 1992.

[Bur84] Warren Burton. Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs. *ACM TOPLAS*, 6(2), 1984.

[CA82] T. Chou and J.A. Abraham. Load Balancing in Distributed Systems. *IEEE Transactions on Software Engeneering*, SE–8(No.4):662–675, Jul 1982.

[CA83] T.C.K. Chou and J.A. Abraham. Load Redistribution Under Failure in Distributed Systems. *IEEE Transactions on Computers*, C–32(9):799–808, Sep 1983.

[CDG+86] M. Castan, M.H. Durand, G.Durrieu, B. Lecussan, and M. Lemaitre. MaRS: a Multiprocessor Machine for Parallel Graph Reduction. *Proc. of the 19th Hawai Int. Conf. on SYSTEM SCIENCES, Honolulu*, pages 152–159, Jan 1986.

[CDL87] M. Castan, M.H. Durand, and M. Lemaitre. A set of combinators for abstraction in linear space. *In Information Processing Letter, North Holland*, 24:183–188, 1987.

[CDM90] E. Cousin, G. Durrieu, and D. Marre. De la gestion des ressources dans une architecture parallèle. Le cas de MaRS. *Actes du 2ème symp. sur les Architectures Nouvelles de Machines, Toulouse*, Sep 1990.

[Chr94] M. Christaller. ATHAPASCAN-0: A control parallelism approach on top of PVM. *In Proc. PVM Users' group meeting, University of Tenessee, Oak ridge, USA*, 1994.

[Col89] M.I Cole. Algorithmic Skeletons: Structured Management of Parallel Computation. *Research Monographs in Parallel and Distributed Computing. Pitman*, 1989.

[Cou91] E. Cousin. *Compilation optimisée d'un langage fonctionnel pour une machine parallèle à réduction de graphe.* PhD thesis, Ecole Nationale Supérieure de L'Aéronotique et de l'Espace, 1991.

[Cou92] L. Courtrai. *Les Composants Actifs de Communication : Outils pour la conception et l'implantation de langages parallèles objets actifs pour machines MIMD.* PhD thesis, Université de LilleI, 1992.

[CS89] A. Calderwood and P. Szeredi. Scheduling Or-Parallelism in Aurora - The Manchester Scheduler. *6$^{th}$ Conf. on Logic Programming, Lisbonne, MIT Press*, Jun 1989.

[CT93] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles.* InterEditions, 1993.

[Den79] J. B. Dennis. The varieties of data-flow computers. *IEEE Transactions on Parallel and Distributed Systems*, pages 430–439, 1979.

[Den96] Y. Denneulin. Utilisation des priorités en contexte distribué pour des applications irrégulières. *Pub. interne AS–171, Séminaires jeunes chercheurs de ParDis, LIFL, Univ. de LilleI*, Mai–Juin 1996.

[Dev90] N. Devesa. *Proposition d'un schéma d'évaluation parallèle du langage fonctionnel FP sur un réseau de processus.* PhD thesis, Université de LilleI, 1990.

[Dow95] S. Dowaji. *Contribution à l'étude des problèmes d'équilibrage de charge dans les environnements distribués.* PhD thesis, Université de Versailles, 1995.

[DR81] J. Darlington and M.J. Reeve. ALICE: A Multiple-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. *In FPCA '81*, pages 65–76, 1981.

[DTM94] A. Diwan, D. Tarditi, and E. Moss. Memory Subsystem Performance of Programs Using Copying Garbage Collection. *In Principles of Programming Languages, ACM*, pages 1–14, 1994.

[ea91] M.L. POWELL et al. SunOS Multi-thread Architecture. *USENIX, Dallas, Texas*, Winter 1991.

[ELZ86] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive Load Sharing In Homogeneous Distributed Systems. *IEEE Transactions on Software Engeneering*, SE–12(No.5):pp. 662–675, 1986.

[ERS90] F. Ercal, J. Ramanujam, and P. Saddayapan. Task allocation onto a hypercube by recursive mincut bipartitioning. *Journal of Parallel and Distributed Computing*, Vol.10(No.1):35–44, Sep 1990.

[Fon94] C. Fonlupt. *Distribution dynamique de données sur machines SIMD.* PhD thesis, Université de LilleI, 1994.

[for94] MPI forum. MPI: A Message Passing Interface standard. Technical report, Apr 1994.

[FR96] Bertil Folliot and Pierre Raverdy. Adaptive Partitionning and Dynamic Allocation For Large Computing Systems. *PDPTA'96 proceedings, Sunnyvale, California, USA*, pages 1268–1279, 9–11 Aug 1996.

[FS94] A. Ferrari and V. Sanderam. A Threads-Based Interface and Subsystem for PVM. *CSTR-940802, University of Virginia and Emory University*, Aug 1994.

[GBea94] A. Geist, A. Beguelin, and J. Dongarra et al., editors. *PVM: Parallel Virtual Machine, A User's guide and tutorial for Networked Parallel Computing.* MIT Press, 1994.

[GBS91] D. Stève G. Bernard and M. Simatic. Placement et migration de processus dans les systèmes répartis faiblement couplés. *TSI 04/91/355-19*, pages 375–392, 1991.

[Gei96] J.M. Geib. Processus légers distribués et régulation de charge. *Ecole Placement Dynamique et Répartition de Charge, France*, pages 89–102, Jul 1996.

[GH86] B. Goldberg and P. Hudak. Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor. *In Workshop on Graph Reduction, Santa Fé, New Mexico, Springer-Verlag LNCS 279*, pages 94–113, Sep 1986.

[GKW85] J.R. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Comm. ACM*, 28(1):34–52, Jan 1985.

[Glo89] F. Glover. Tabu search - Part I. *ORSA Journal of Computing*, 1(3):190–206, 1989.

[Gol88] B.F. Goldberg. Multiprocessor Execution of Functional Programs. *Intl. Journal of Programming*, 17(5):425–473, 1988.

[GS90] B. Gavish and O.R. Liu Sheng. Dynamic File Migration in Distributed Computer Systems. *Comm. ACM*, 33(2), Feb 1990.

[Ham94] K. Hammond. Parallel Functional Programming: An Introduction. *Proc. First Intl. Symp. on Parallel Symbolic Computation, World Scientific Publishing Company*, Sep 1994.

[HCG+82] K. Hwang, W.J. Croft, G.H. Goble, B.W. Wah, F.A. Briggs, W.R. Simons, and C.L. Coates. A Unix-Based Local Computer Network with Load Balancing. *IEEE Computer*, Apr 1982.

[Hem94] F. Hemery. *Etude de la répartition dynamique d'activités sur architectures décentralisées.* PhD thesis, Université de LilleI, 1994.

[HG84] P. Hudak and B. Goldberg. Experiments in Diffused Combinator Reduction. *Proc. of the ACM Symp. on Lisp and Functional Programming*, 1984.

[HG85] P. Hudak and B. Goldberg. Serial Combinators: Optimal Grains of Parallelism. *FPCA*, Springer-Verlag LNCS(210):382–388, Sep 1985.

[HJ84] A. Hac and T.J. Johnson. Sensitivity study of the load balancing algorithm in a distributed system. *Journal of Parallel and Distributed Computing*, 10(1):85–89, 1984.

[HJ92] K. Hammond and S.L. Peyton Jones. Profiling Scheduling Strategies on the GRIP Multiprocessor. *Proc. 4th Int. Workshop on Parallel Implementation of Functional Languages, H. Küchen (ed.), Aachen University*, Sep 1992.

[HJ95] M. Hamelin and J. Julliand. Compilation d'un langage fonctionnel basé sur des flots de données en PVM. *Workshop on Parallel and Distributed Computing, Biel*, Oct 1995.

[HJ96] M. Hamelin and J. Julliand. Execution of parallel functional language using PVM. *Proc. of the 4th IASTED Int. Conf., Innsbruck, Austria*, Fev 1996.

[HS86] P. Hudak and L. Smith. Para-functional Programming: A Paradigm for Programming Multiprocessor Systems. *In ACM POPL*, pages 243–254, Jan 1986.

[HTG95] Z. Hafidi, E.G. Talbi, and G. Goncalves. Load balancing and parallel tree search: The MPIDA* algorithm. *Parco'95 proc. Gent Belgium*, Sept 1995.

[HTG96] Z. Hafidi, E-G. Talbi, and J-M. Geib. MARS: Adaptive scheduling of parallel applications in a multi-user heterogeneous environment. *ESPPE'96 proceedings, Alpe d'Huez, France*, pages 119–122, Apr 1996.

[Hug82] J. Hughes. Supercombinators: a new implementation method for applicative languages. *In Prentice Hall Int. Series in Computer Science, editor, ACM Conference on LISP and Functional Programming*, pages 1–10, Aug 1982.

[JCSH87] S.L. Peyton Jones, C. Clark, J. Salkild, and M. Hardie. GRIP: a High-Performance Architecture for Parallel Graph Reduction. *In FPCA'87, Springer-Verlag LNCS 274*, pages 98–112, Sep 1987.

[JH93] M.P. Jones and P. Hudak. Implicit and Explicit Parallel Programming in Haskell. *Research Report YALEU/DCS/RR-982, Department of Computer Science, Yale Univ., New Haven*, Aug 1993.

[Joh83] T. Johnsson. The G-machine: An abstract machine for graph reduction. *Declarative Programming Workshop, University College London*, pages 1–20, Apr 1983.

[Jon87] S.L. Peyton Jones. *The implementation of functional programming languages.* Prentice Hall International series in Computer Science, 1987.

[Jon89] S.L. Peyton Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, Vol. 32(No. 2):175–186, Apr 1989.

[JS89] S.L. Peyton Jones and J. Salkild. The Spinless Tagless G-machine. *In Proc. FPCA'89, London, ed MacQueen, Addison Wesley*, 1989.

[Juu90] N.C. Juul. A distributed garbage collector for emerald. *In ECOOP/OOPSLA Workshop on Garbage Collection in Object-Oriented Systems, Ottawa, Canada*, 1990.

[Kea94] J.A. Keane. An overview of the Flagship system. *J. of Functional Programming*, 4(1):19–45, Jan 1994.

[Kel89] P. Kelly. Functional Programming for Loosely-coupled Multiprocessors. *Research Monographs in Parallel and Distributed Computing. Pitman*, 1989.

[KL84] R.M. Keller and F.C.H. Lin. Simulated performance of a reduction-based multiprocessor. *IEEE Transactions on Software Engeneering*, 17(7), 1984.

[Kor85] R. E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, Vol. 32(No. 27):pp. 97–109, Feb 1985.

[KW91] H. Kuchen and A. Wagener. Comparison of Dynamic Load Balancing Strategies. *K. Boyanov ed., Parallel and Distributed Processing, North-Holland*, pages 303–314, Mar 1991.

[LB89] D. Lester and G.L. Burn. An Executable specification of the HDG-machine. *Proc. Workshop on Massive Parallelism: Hardware, Programming and Applications, Italy*, Oct 1989.

[Les89] D. Lester. An Efficient Distributed Garbage Collection Algorithm. *Proc. PARLE'89, Springer-Verlag LNCS 365*, pages 207–223, jun 1989.

[LK90] F.C.H. Lin and R.M. Keller. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engeneering*, SE–13(No.1):32–38, Jan 1990.

[Lo84] V.M. Lo. Heuristics algorithms for task assignment in distributed systems. *Proc. of the int. Conf. on Distributed Computing Systems, San Francisco*, pages 30–39, 1984.

[LR92] Hwa-Chun Lin and C. S. Raghavendra. A Dynamic Load-Balancing Policy With a Central Job Dispatcher (LBC). *IEEE Transactions on Software Engeneering*, 18(2):148–158, Feb 1992.

[LS88] I. Lee and D. Smitley. A synthesis algorithm for reconfigurable interconnexion networks. *IEEE Transactions on Computers*, Vol.C–37(No.6):691–699, Juin 1988.

[MA87] C.E. McDowell and W.F. Appelbe. Nearest-Neighbor mapping of finite element graphs onto processor meshes. *IEEE Transactions on Computers*, Vol.C–36(No.12):1408–1424, Dec 1987.

[Mat93] J.S. Mattson. *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction.* PhD thesis, University of California, San Diego, 1993.

[MDLT96a] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. An Adaptive Load Balancing Algorithm with a Multithreaded Implementation. *Proc. of the 11$^{th}$ Int. Conf. On Systems Engineering (ICSE'96), University of Nevada, Las Vegas, USA*, pages 97–102, 9–11 Jul 1996.

[MDLT96b] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. Adaptive Load Balancing and Irregular Applications. *Publication interne AS-173, LIFL, Université de LilleI*, Jun 1996.

[MDLT96c] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. Adaptive Load Balancing and Multithreading. *Proc. of the 9$^{th}$ ISCA Intl. Conf. on Parallel and Distributed Computing Systems, PDCS'96, Dijon, FRANCE*, 1:343–348, 25–27 Sep 1996.

[MDLT96d] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. Adaptive load balancing of irregular applications. A case study: IDA* applied to the 15-puzzle problem. *Springer-Verlag LNCS 1117, Proc. of the Third Intl. Workshop, IRREGULAR'96, Santa Barbara, California, USA*, pages 327–338, 19–21 Aug 1996.

[MDLT96e] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. An Adaptive Load Information Collection Policy. A case study: Dynamic Load Balancing. *Publication interne AS-172, LIFL, Université de LilleI*, May 1996.

[MDLT96f] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. An Adaptive Load Information Collection Policy. *PDPTA'96 proceedings, Sunnyvale, California, USA*, pages 649–658, 9–11 Aug 1996.

[MDLT96g] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. A multithreaded system for adaptive load balancing in distributed and parallel architectures. *Proc. of the 2$^{nd}$ European School of Computer Science (ESPPE'96), Ecole Alpe d'Huez, Grenoble, FRANCE*, pages 129–132, 1–5 Apr 1996.

[MDLT96h] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. Un système adaptatif et multi-threads de collecte d'informations de charge. Application à la régulation de charge. *Poster, Actes des 8$^{es}$ Rencontres Francophones du Parallélisme (RenPar'8), Bordeaux, France*, 20-24 mai 1996.

[Mel92] N. Melab. Bokareg : Un algorithme heuristique pour le placement statique de tâches sur un réseau de transputers. *Rapport de DEA, LIFL/USTL*, juil 1992.

[Mel94] N. Melab. Synthèse des méthodes de distribution statique et dynamique de la charge sur architectures MIMD. Etude de cas : langages fonctionnels. *Publication interne. Univ. de Lille I, LIFL*, AS–160, Oct 1994.

[MG95] T. Monteil and J.M. Garcia. Network-analyser : Un système de collecte et de prédiction de l'état d'un réseau local pour le placement dynamique de processus. *CAPA*, pages 81–84, Apr 1995.

[MKH91] E. Mohr, D.A. Kranz, and R.H. Halstead. Lazy Task Creation - a Technique for Increasing the Granularity of Parallel Programs. *ACM Conf on Lisp & Functional languages*, 2(3):264–280, Jul 1991.

[MLT82] P.R. Ma, E.Y. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, Vol.C–31(No.1):41–47, Jan 1982.

[MM89] V.F. Magirou and J.Z. Milis. An algorithm for the multiprocessor assignment problem. *Operations Research Letters 8*, pages 351–356, Dec 1989.

[MS87] D.L. McBurney and M.R. Sleep. Transputer-Based Experiments with the ZAPP Architecture. *In PARLE'87*, Springer-Verlag 258:242–259, 1987.

[MS89] G.A. Mago and D.F. Stanat. The FFP Machine. *In High-Level Language Computer Architectures*, pages 430–468, 1989.

[MT91] T. Muntean and E-G. Talbi. Méthodes de placement statique de processus sur architectures parallèles. *TSI 04/91/355-19*, 1991.

[Nam97] Raymond Namyst. *$PM^2$ : Un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières sur architectures distribuées.* PhD thesis, Université de LilleI, 1997.

[Nil80] N.J. Nilsson. Principles of artificial intelligence. *Palo Alto, CA:Tioga*, 1980.

[NM95] R. Namyst and J.F. Méhaut. $PM^2$: Parallel Multithreaded Machine. A computing environment for distributed architectures. *North Holland, Parco'95 proc. Gent Belgium*, pages 279–285, Sept 1995.

[NPA92] R.S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. *In 19$^t$h ACM Annual Symp. on Comp. Arch.*, (1):156–167, 1992.

[NXG86] L.M. Ni, C-W. Xu, and T.B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engeneering*, Vol. C–36(No.10):pp 1153–1161, Oct 1986.

[Par91] A.S. Partridge. *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages.* PhD thesis, University of Tasmania, 1991.

[PC84] C.C. Price and S. Crishnaprasad. Software allocation models for distributed computing systems. *Proc. of the 4th Int. Conf. on Distributed Computing Systems, San Francisco, California*, pages 40–48, Mai 1984.

[pPS89] Edité par Perihelion Software. *The HELIOS Operating System.* Prentice-Hall, 1989.

[PPTV91] B. Plateau, P.Raynal, D. Trystram, and F. Vincent. Placement de tâches : un tour d'horizon des techniques efficaces. *Rapport technique IMAG*, Juin 1991.

[PTS88] S. Pulidas, D. Towsley, and J.A. Stankovic. Imbedding gradient estimators in load balancing algorithms. *In Proc. of the 8$^{th}$ Int. Conf. On Distributed Computing Systems, San Jose, California,*, pages 482–490, 1988.

[Rab91a] F.A. Rabhi. Divide-and-Conquer and parallel graph reduction. *Parallel Computing, North Holland, Amsterdam*, 1991.

[Rab91b] F.A. Rabhi. Dynamic Combinators: Run-Time Control of the Granularity in Functional Programs. *Department of Computer Science, University of Hull*, 1991.

[Rab93] F.A. Rabhi. Exploiting Parallelism in Functional Languages: A Paradigm Oriented Approach. *In T. Lake and P. Dew, editors, Abstract Machine Models for Highly Parallel Computers, Oxford Univ. Press*, 1993.

[Rav95] P.G. Raverdy. Une Politique Extensible de Partitionnement pour les Environnements Parallèles et Répartis. *CAPA*, pages 9–11, Apr 1995.

[RL91] E. Reiher and G. Lapalme. Réduction de graphe parallèle et spéculative d'un langage fonctionnel. *BIGRE 72*, pages 89–101, Jan 1991.

[RM91] F.A. Rabhi and G.A. Manson. Experiments with a transputer-based parallel graph reduction machine. *CONCURRENCY: PRACTICE AND EXPERIENCE*, 3(4):413–422, Aug 1991.

[RS87] C.A. Ruggiero and J. Sargeant. Control of Parallelism in the Manchester Dataflow Machine. *In FPCA '87, Springer-Verlag LNCS 274*, pages 1–15, 1987.

[San94] N. Sankaran. A Bibliography on Gargage Collection. Technical Report 94-102, Clemson University, Departement of Computer Science, Fev 1994.

[Sar87] J. Sargeant. Load Balancing, Locality, and Parallelism Control in Fine-Grain Parallel Machines. *Tech. Report UMCS-86-11-5, Manchester University*, 1987.

[Sch93] W. Schreiner. Parallel Functional Programming (An annotated bibliography). Technical report, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, May 1993.

[SE86] P. Saddayapan and F. Ercal. Processor scheduling for linearly connected parallel processors. *IEEE Transactions on Computers*, Vol.C–35(No.7):632–638, Jul 1986.

[Sin87] J.B. Sinclair. Efficient computation of optimal assignments for distributed tasks. *Journal of Parallel and Distributed Computing*, Vol.4:342–362, 1987.

[SJ93] P.M. Sansom and S.L. Peyton Jones. Generational garbage collection for Haskell. *In FPCA, ACM*, Jun 1993.

[SS84] J.A. Stankovic and I.S. Sidhu. An Adaptive Bidding Algorithm For Processes, Clusters and Distributed Groups. *Proc. 4th Int. Conf. On Distributed Computing Systems, San Francisco*, pages 49–59, Mai 1984.

[ST85] C-C. Shen and W-H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minmax criterion. *IEEE Transactions on Computers*, Vol.C–34(No.3):197–203, Mar 1985.

[Sto77] H.S. Stone. Program assignment in three–processor systems and tricutset partitioning of graphs. *Tech. Rep. No.ECE–CS–77–7, Dep. of Elect. & Computer Eng., Univ. of Massachussets, Amherst*, 1977.

[Tal95] E-G. Talbi. Allocation dynamique de processus dans les systèmes distribués et parallèles : Etat de l'art. *Publication interne. Univ. de Lille I, LIFL*, AS–162, Jan 1995.

[TB91] E-G. Talbi and P. Bessiere. Un algorithme génétique massivement parallèle pour le problème de partitionnement de graphes. *Rapport technique LGI–IMAG*, 1991.

[THM+96] P.W. Trinder, K. Hammond, J.S. Mattson, A.S. Partridge, and S.L. Peyton Jones. GUM: a portable parallel implementation of Haskell. *In Proc. of Programming Language Design and Implementation, Philadelphia, USA*, May 1996.

[TL89] M. M. Theimer and K.A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engeneering*, 15(1), Nov 1989.

[TM90] E-G. Talbi and T. Muntean. Placement statique de processus sur une architecture parallèle. *Rapport de recherche LGI–IMAG*, RR 833–I–, Nov 1990.

[WH87] P. Wadler and R.J.M. Hughes. Projections for strictness analysis. *FPCA '87, Springer-Verlag LNCS 274*, pages 385–407, 1987.

[WLM92] P.R. Wilson, M.S. Lam, and T.G. Moher. Caching considerations for generational garbage collection. *In SIGPLAN Symposium on Lisp and Functional Programming, San Francisco, California*, 1992.

[WM85] Y.T. Wang and J.T. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, C–34(No.3):204–217, Mar 1985.

[WSWW87] I. Watson, J. Sargeant, P. Watson, and J.V. Woods. Flagship Computational Models and Machine Architecture. *In Int. Computers Ltd. Technical Journal*, 1987.

[WW87] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. *PARLE'87*, Springer-Verlag LNCS 259:432–443, 1987.

[WY92] W.F. Wong and C.K. Yuen. A Model of Speculative Parallelism. *In Parallel Processing Letters*, 2(3):265–272, 1992.

[XH91] J. Xu and K. Hwang. Heuristic Methods for Dynamic Load Balancing in A Message-Passing Supercomputer. *IEEE*, pages 888–897, Apr 1991.