HAL
open science

Placement de taches sur ordinateurs paralleles a memoire distribuee

Pascal Bouvry

To cite this version:

Pascal Bouvry. Placement de taches sur ordinateurs paralleles a memoire distribuee. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1994. Français. NNT : . tel-00005081

HAL Id: tel-00005081

https://theses.hal.science/tel-00005081v1

Submitted on 25 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée par

Pascal BOUVRY

pour obtenir le titre de DOCTEUR

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(Arrêté ministériel du 30 Mars 1992)

Spécialité : **Informatique**

Placement de tâches sur ordinateurs parallèles à mémoire distribuée

---

Date de soutenance : 6 octobre 1994

Composition du jury :

Président	Jacques	MOSSIERE	Prof. ENSIMAG
Rapporteurs	Jacek	BLAZEWICZ	Prof. Univ. de Poznan
	Jean-Claude	KONIG	Prof. Univ. d'Evry
Examinateurs	Jacques	CHASSIN de KERGOMMEAUX	CR CNRS LGI-IMAG
	Yves	SOREL	DR INRIA-Rocquencourt
	Denis	TRYSTRAM	Prof. ENSGI

Thèse préparée au sein du
Laboratoire de Modélisation et de Calcul
de l'Institut d'Informatique et de Mathématiques Appliquées de Grenoble

Remerciements

Je tiens tout d'abord à remercier les personnes suivantes :

Les membres du jury

• Jacques Mossière (BULL-IMAG), professeur à l'ENSIMAG, de m'avoir fait l'honneur de présider ce jury.
• Denis Trystram (LMC-IMAG), professeur à l'ENSGI, pour m'avoir proposé de faire cette thèse et pour avoir dirigé mes recherches tout au long de ces trois années qui m'ont semblées bien courtes.
• Jacek Błażewizc, professeur à l'Université de Poznan (Pologne), d'avoir bien voulu être rapporteur de cette thèse. Je tiens aussi à remercier Jacek Błażewicz de la qualité de son enseignement et du travail que nous avons pu faire ensemble.
• Jean-Claude Konig, professeur à l'Université d'Evry d'avoir bien voulu être rapporteur de cette thèse. Je tiens aussi à remercier Jean-Claude Konig de ses précieuses remarques m'ayant permis d'améliorer la présente thèse et aussi sa soutenance.
• Yves Sorel, directeur de recherche à l'INRIA-Rocquencourt, de l'intérêt qu'il a montré face à la présente thèse et de sa présence dans le jury.
• Jacques Chassin de Kergommeaux (LGI-IMAG), chargé de recherche au CNRS, pour avoir bien voulu être examinateur de cette thèse et m'avoir fait participer au projet européen SEPP (Software Engineering for Parallel Processing).

Les gens qui m'ont aidé à la réaliser

• Le ministère de la Recherche et de l'Enseignement Supérieur français qui a financé ces trois années de recherche.
• Joao Paulo Kitajima (LGI-IMAG) avec qui j'ai eu vraiment plaisir à travailler sur la partie évaluation de performances.
• Brigitte Plateau (LGI-IMAG), responsable du projet APACHE, projet INRIA, dont le présent travail fait partie.
• Les nombreuses réunions remue-méninges de l'équipe d'évaluation de performances (LGI-IMAG) et l'équipe Apache, avec entre autres, Jacques Briat, Brigitte Plateau, Jean-Louis Roch, Jean-Marc Vincent, Eric Maillet, Alain Fagot et Cécile Tron.
• L'équipe polonaise de Poznan dirigée par Jacek Błażewizc et en particulier à Rafal Walkowiak (bonjour aussi à Mace et à Janucz).
• Les nombreuses personnes ayant permis d'étendre ce travail dont Cécile Tron pour ses fonctions de coûts prenant en compte la charge réseau ; Ricardo Correa pour l'implémentation d'un *Branch&Bound* parallèle.
• Mes relecteurs dont : Christophe Calvin, Jacques Chassin de Kergommeaux,

Thierry Gautier, Frédéric Guinand, Stéphane Ricour et les membres du jury.
• Les docteurs Colombet, Guinand, Michallon, Gautier et maîtres Sauze, Ricour pour les longues réflexions sur l'existence et la soif d'absolu dans la quête scientifique. En n'oubliant pas non plus l'ex-tôlard (centralien) François Vincent.
• Les gens m'ayant aidé à réaliser le pot d'après thèse : Inès pour son clafoutti, Nathalie pour sa mousse au chocolat et Stéphane pour l'arrivage de pralines.

Les gens ayant influencé mon avenir d'informaticien

• Je tiens à remercier Camille Cambier de m'avoir initié à ce virus qu'est l'informatique.
• Je tiens à remercier Baudouin Le Charlier d'avoir bien voulu me faire confiance en dirigeant mon mémoire de maîtrise.
• Je tiens à remercier Jean Della Dora d'avoir eu une ouverture d'esprit suffisamment large que pour s'intéresser aux stagiaires des pays voisins.
• Je tiens à remercier Jean-Louis Roch, d'avoir bien voulu superviser mon stage de maîtrise au LMC-IMAG.
• Je tiens à remercier Denis Trystram de m'avoir un jour branché sur le domaine du placement de tâches et de m'avoir pris en thèse.
• Je tiens à remercier Jacques Chassin de m'avoir prévenu que le CWI cherchait un post-doc. Je tiens aussi à remercier Paul ten Hagen et Farhad Arbab du CWI de m'avoir proposé la position post-doctorale à Amsterdam où je suis en rédigeant ces lignes.

Les gens faisant partie de mon entourage

• Petit gars, guruji, un jour je suivrai ta voie et on fondera une communauté dans le Jura. Lolo, Phil, Roland, Fred, Titou, Eric et Simone, je compte sur vous pour fonder la filiale française pendant mon exil.
• Mes amis et collègues français (et luxembourgeois) ayant agrémenté ces trois années et demi : Cécile, Claire, Ines, Isabelle, Evelyne, Françoise, Joëlle, Nathalie(s), Simone, Alain(s), Christophe, Dédé, Dominique, Fambon, Fred, Gilles, Jean-Yves, Lolo, Michel, Nounouille, Phil, Roland, Titou, Yannick, ...
• Mes amis belges dont : Catherine, Dominique, Gene, Karine, Bast, B17, Bernard, Carmin, Eric, F242, Fred, Jean, J-F, Michel, Olivier(s), Philippe, Roland, Stéphane, Toni, Vincent, Yves, ...

Ma famille

• Mon père, ma mère et ma grand-mère qui m'ont activement (et financièrement) supporté durant toutes ces années d'études. Je tiens aussi à remercier mon frangin, sans qui on s'embêterait à la maison.

Cette thèse est dédiée à mon grand-père

Table des matières

1 Introduction **13**
1.1 Méthodologie 14
1.2 Plan de la thèse 15
1.3 Originalité du travail 16

2 Modèle de machines **19**
2.1 Introduction 19
2.2 Machine séquentielle 19
2.2.1 Processeurs 19
2.2.2 Mémoire 21
2.2.3 Les réseaux 22
2.3 Machine parallèle à mémoire distribuée 23
2.3.1 Les unités de traitement 23
2.3.2 Le réseau d'interconnexion 24
2.3.3 Connexion vers l'extérieur 26
2.4 Paramètres matériels 27
2.5 Objectif 28
2.6 Conclusion 29

3 Modèle de programmation **31**
3.1 Introduction 31
3.2 Environnement 32
3.3 Système d'exploitation 34
3.3.1 Services offerts sur les nœuds 34
3.3.2 Services offerts entre les nœuds 36
3.3.3 Utilisation d'un processeur 37
3.4 Modèles de programmes 38
3.4.1 Paramètres logiciels 38
3.4.2 Graphe de précédence 39
3.4.3 Graphes de tâches sans précédence 40
3.4.4 Autres modèles 42
3.5 Conclusion 42

4 Résultats existants **45**
4.1 Introduction 45
4.2 Chaîne de compilation 46
4.3 Ordonnancement/regroupement 48
4.3.1 Définitions et objectifs 48
4.3.2 Résultats d'ordonnancement 49
4.3.3 Algorithmes de regroupement 50
4.3.4 Avantages/désavantages 51
4.4 Placement 52
4.4.1 Critères de placement 53
4.4.2 Les différentes solutions 55
4.4.3 Autres solutions et conclusion 64

5 Une plate-forme d'aide au placement **67**
5.1 Introduction 67
5.2 Objectif 67
5.3 Modèle de machine 70
5.4 Modèle de programme 70
5.5 Fonction objective 71
5.6 Spécification des algorithmes implémentés 72
5.6.1 les gloutons 72
5.6.2 Les algorithmes itératifs 74
5.6.3 Les algorithmes exacts 81
5.7 Solution proposée 82
5.7.1 Intégration à un environnement de programmation 85
5.7.2 Description des outils de prise de traces 86
5.8 Prise en compte de contraintes supplémentaires 86
5.9 Conclusion 87

6 Validation **89**
6.1 Introduction 89
6.2 Graphes aléatoires 89
6.3 ANDES: Un outil d'évaluation de performances 92
6.3.1 L'approche graphes synthétiques 92
6.3.2 Le principe de *ANDES-Synth* 94
6.3.3 Les objectifs 96
6.3.4 Le jeu de tests 96
6.3.5 Les modèles de machines 97
6.3.6 Tests effectués 100
6.4 Analyse des résultats 101
6.4.1 Représentativité des tests 101
6.4.2 Adéquation de la fonction objective 102
6.4.3 Amélioration apportée 105

6.4.4 Performance des différents algorithmes 105
6.5 Itération 107
6.6 Conclusion 108

7 Conclusion et perspectives 109
7.1 Conclusion 109
7.2 Perspectives 111

A Description des tests 113

B Récapitulatifs des tests 125

C Génie logiciel 131
C.1 Introduction 131
C.2 Interfaces 131
C.2.1 Langage de description de graphes 131
C.2.2 Interface X-Window 132
C.2.3 Table de routages 133
C.3 Fonctionnement interne 133
C.3.1 Bibliothèques 133
C.3.2 Gestion de matrices creuses 134

Table des figures

2.1 Architecture du RS6000 20
2.2 Hiérarchie des mémoires 22
2.3 Schéma synoptique du transputer T800 23
2.4 Tore tronqué constitué de transputers 25
2.5 SP2 d'IBM avec 16 nœuds 26

3.1 Environnement de programmation parallèle 32
3.2 Découpe en couches du système de communications 36
3.3 Exemple de graphe de précédence 39
3.4 Exemple de graphe de tâches 40

4.1 Chaîne de compilation 46
4.2 Détail du module placement/ordonnancement 47
4.3 Regroupement et diagramme de GANTT 51

5.1 Deux instances de OUF 69
5.2 Appel de service et découpe en couches de Athapascan 69
5.3 Evolution du placement des différentes tâches de l'algorithme de Strassen par LPTF et LGCF. 74
5.4 Comparaison entre descente de température conseillée par la théorie et par la pratique 75
5.5 Acceptation ou rejet d'un mouvement de coût $\delta_{f_{ij}}$ 76
5.6 Comparaison entre température initiale et qualité de solution . . . 77
5.7 Recuit simulé 78
5.8 Evolution de la fonction de coût pour une recherche tabu 80
5.9 Comparaison de deux tailles de liste tabu 80
5.10 Arbre de recherche du A* 81
5.11 Intégration de l'outil de placement et de Pyrros. 83
5.12 Amélioration du placement après exécution 84
5.13 Intégration de l'outil de placement sur le Méganode 85

6.1 Comparaison sur graphes aléatoires. 91
6.2 L'environnement *ANDES-Synth*. 95
6.3 ACP : Représentation des individus (tests) 102

6.4 Fonctions de coût et temps d'exécution 104
6.5 Comparaison entre tabu et LGCF 107
6.6 Améliorations successives d'un placement 108

A.1 La structure de BELLFORD2. 113
A.2 Les structures de DIAMOND1 et de DIAMOND2. 114
A.3 La structure de DIAMOND3. 115
A.4 La structure de DIAMOND4. 115
A.5 La structure de DIVCONQ. 116
A.6 La structure de FFT2. 116
A.7 La structure de GAUSS 117
A.8 La structure de ITERATIVE2. 117
A.9 La structure de MS3. 118
A.10 La structure de PDE1. 119
A.11 La structure de PDE2. 120
A.12 La structure de PROLOG. 120
A.13 La structure de QCD2. 121
A.14 La structure de STRASSEN. 122
A.15 La structure de WARSHALL. 123

C.1 Interface X-Window 132
C.2 Choix d'un fichier à éditer 133
C.3 Gestion de matrices 134

Chapitre 1

Introduction

Pour répondre à la demande croissante en besoins de calcul, différentes méthodes ont été proposées. Les ordinateurs séquentiels classiques continuent à augmenter régulièrement leur vitesse, mais les limites de la physique font que d'ici quelques années, leur évolution devrait stagner. Certains chercheurs examinent les solutions physiques à y apporter telles que les supra-conducteurs, les ordinateurs optiques, etc. Une autre voie qui est explorée, entre autres, dans cette thèse, consiste à utiliser les technologies existantes et à les assembler afin de former des ordinateurs parallèles. Remarquons que les deux solutions ne sont pas opposées et que les solutions matérielles pourront intégrer les ordinateurs parallèles de demain qui ne seront que plus puissants ...

Différentes catégories de machines parallèles, constituées d'un ensemble de nœuds de travail, existent actuellement sur le marché[Fly66]. Parmi celles-ci, on peut retrouver les machines SIMD [1] (par exemple la Maspar, et la CM2 de TMC). Ces machines exécutent à chaque cycle de leur horloge globale une même instruction sur chaque processeur ; cette instruction s'appliquant à des données différentes. Une deuxième catégorie comprend les machines MIMD [2] où chaque nœud possède son propre programme et travaille sur ses propres données. Certaines de celles-ci possèdent une mémoire partagée entre tous les nœuds (la gestion des accès concurrents à la mémoire limite le nombre de processeurs de ce type de machine), tandis que d'autres ont une mémoire distribuée. Ce dernier type de machine, MIMD à mémoire distribuée, est le type de machine qui nous intéressera tout au long de cette thèse. Ceci principalement pour deux raisons : une première est la curiosité scientifique, en effet ce domaine est encore largement inexploré, et une seconde plus matérialiste est de constater que ce paradigme commence à s'imposer à la majorité des constructeurs (SP1 d'IBM, CS2 de PCI, Paragon d'Intel, CM5 de TMC, T3D de Cray).

Les nombreuses recherches effectuées dans le cadre des machines SIMD ont abouti à des techniques de programmation connues et validées. Ces techniques,

[1] SIMD: *Single Instruction Multiple Data*

[2] MIMD: *Multiple Instructions Multiple Data*

après adaptation pour gérer l'asynchronisme, sont aujourd'hui aussi utilisées sur les machines MIMD sous l'appellation SPMD [3]. Des techniques de programmation propres aux machines MIMD sont développées actuellement afin de tirer, avec difficulté il est vrai, parti du maximum de possibilités des machines MIMD. C'est ce type de programmation qui est au centre de cette thèse et nous nous intéressons plus particulièrement à la catégorie de programmes parallèles pouvant être décrits de manière statique sous forme d'un graphe de tâches[CD72]. Les tâches qui constituent les sommets du graphes sont, dans ce modèle,des boîtes noires caractérisées par un coût de calcul et une série de connexions vers d'autres tâches. Ces connexions sont caractérisées par un coût communication et constituent les arêtes du graphe. Le but du placement est d'assigner à chacune des tâches un processeur de manière à optimiser certains objectifs dont le plus utilisé est la minimisation du temps total d'exécution du programme[NT93][ERL90]. La description du programme est faite à gros grains, c'est-à-dire que la taille du graphe de tâches n'est pas trop grande ($\leq 10^4$). Le but de cette thèse consiste à étudier le problème du placement des tâches sur les processeurs et à fournir un environnement d'aide au placement (automatique ou semi-automatique). Ceci comprend donc une partie théorique qui consiste à étudier les modèles de programmes, de machines et les résultats existants. Cette partie comprend aussi la conception d'une méthode de résolution, la conception et l'implémentation d'algorithmes la mettant en œuvre ainsi que la validation de ces derniers. De nombreux aspects pratiques tels que l'implémentation de l'outil de placement, l'interfaçage de celui-ci avec un environnement de programmation constituent aussi une partie importante de cette thèse.

1.1 Méthodologie

Afin de permettre de réaliser la conception et l'implémentation d'un outil d'aide au placement, différentes étapes sont nécessaires :

1. La première étape consiste à définir et spécifier plus précisément les différents modèles de machine et de programmation utilisés. Ceci comprend une étude de l'architecture des machines cibles, une étude des différentes couches logicielles et des langages utilisés sur ces machines.

2. La deuxième étape consiste à étudier les solutions proposées au problème du placement dans la littérature. Elle consiste de plus à étudier les algorithmes utilisés dans la résolution de problèmes analogues et leur adéquation au problème du placement.

[3] SPMD: *Single Procedure Multiple Data*

3. A partir de tout cela, une solution originale est proposée. Cette solution doit avoir plusieurs qualités :
   - Elle doit être suffisamment générale pour pouvoir s'adapter facilement à différentes machines cibles et être affinée suite à une phase de test.
   - Cette solution doit être effective et donc disposer d'outils de description de graphes et être interfacée à des outils existants.
4. Afin de valider les possibilités des différents algorithmes implémentés une série de tests sont effectués sur des graphes générés aléatoirement.
   Afin de compléter cette première validation, une étude des programmes existants est effectuée afin d'en tirer une série de familles représentatives des programmes réellement utilisés. La phase d'expérimentation/validation a différents objectifs :
   - Montrer l'adéquation entre la méthode utilisée et le but recherché.
   - Différencier les différents algorithmes utilisés.
   - Différencier de manière effective et par rapport au problème du placement, les différents tests.
   - Permettre, en mettant en évidence certains points critiques, l'amélioration du placement et des algorithmes de placement.

Le plan de la thèse suit la méthodologie décrite.

1.2 Plan de la thèse

Dans le deuxième chapitre, je décris les caractéristiques des machines MIMD à mémoire distribuée existantes. Ce chapitre montrera la complexité des processeurs actuels. Une série de paramètres destinés aux problèmes de placement sont extraits de la masse des paramètres caractérisant les machines. Le but à atteindre en spécifiant ces paramètres est de trouver un compromis entre recherche de performances et portabilité. Il s'agit aussi de ne pas énumérer l'ensemble des paramètres qui seront masqués par les différentes couches logicielles et par la granularité assez forte qui est choisie.

Dans le troisième chapitre, dédié aux composantes logicielles, les systèmes d'exploitation des machines parallèles actuelles sont décrits et les différents paramètres représentants les possibilités des systèmes sont mis en évidence. Ces paramètres vont permettre de définir une série de modèles de description de programmes parallèles. Parmi ceux-ci, deux modèles sont décrits plus particulièrement : les graphes acycliques de précédence (DAG) et les graphes de tâches sans précédence.

Le quatrième chapitre est entièrement consacré à un tour d'horizon des solutions apportées au problème de l'ordonnancement et de manière plus approfondie au problème du placement de graphes sans relation de précédence trouvées dans la littérature. L'accent est mis sur l'intégration d'un procédé de placement automatique dans un environnement de programmation parallèle. La double possibilité d'utilisation des algorithmes de placement est soulignée : si le graphe de départ est sans précédence, ils peuvent être utilisés directement tandis que si le graphe de départ est un DAG, une première phase de regroupement (*clustering*) doit être effectuée.

Dans le cinquième chapitre une solution originale est proposée, cette solution s'appuie et est interfacée sur un environnement de programmation complet. Trois types d'algorithmes (gloutons, itératifs et exacts) ont été conçus et implémentés. Parmi ceux-ci, on retrouve plus particulièrement un recuit simulé et une recherche tabu. Ces algorithmes optimisent différentes fonctions objectives (des plus simples et universelles aux plus complexes et ciblées). Dans le cas où le graphe de tâches comporte un grand nombre de nœuds et où l'algorithme placé n'est pas destiné à être utilisé un grand nombre de fois et n'est pas très coûteux, des algorithmes gloutons simples sont à envisager. Tandis que si le graphe de tâches comporte un petit nombre de nœuds (i.e. ≤ 16), des algorithmes exacts peuvent être utilisés. Dans le cas général la solution proposée consiste à effectuer un premier placement à l'aide, par exemple d'un algorithme glouton, et à affiner celui-ci après une première exécution, en effet à ce moment là, des outils de prises de mesures auront permis de mieux connaître les différents paramètres (entre autres, les coûts de calcul et de communication).

Les outils de prise de traces permettront dans le chapitre suivant de valider les différentes fonctions de coût et les différents algorithmes d'optimisation. Un jeu de tests est défini, décrit et utilisé. Les tests sont effectué sur le méganode (machine à 128 transputers), en utilisant comme routeur VCR de Southampton, les outils de génération de graphes synthétiques ANDES du projet ALPES (développé par l'équipe d'évaluation de performances du LGI-IMAG)[KP92] [BKPT94], l'algorithme de regroupement DSC [4] de PYRROS (développé par Tao Yang et Apostolos Gerasoulis)[YG92].

1.3 Originalité du travail

L'originalité de cette thèse réside donc dans une recherche théorique au niveau des modèles de programmation parallèle sur machines à mémoire distribuée. Cette recherche amène à cerner une série de paramètres logiciels et matériels permettant de décrire les différents modèles. Le modèle de machine choisi est MIMD à mémoire distribuée et le modèle de programme choisi est le graphe de tâches sans précédence. Deux utilisations de ce modèle sont envisagées : soit comme un

[4] DSC: Dominant Sequence Algorithm

modèle à part entière, soit comme un modèle intermédiaire dans le cadre du problème du regoupement/placement, ce qui permet de prendre en considération les graphes de précédence. Dans ce dernier cas, le regroupement des tâches est effectué à l'aide de PYRROS (ce qui est équivalent à effectuer un placement sur une architecture virtuelle composée d'un nombre illimité de processeurs) et le placement des groupes sur un architecture réelle est effectué à l'aide de nos outils. La complémentarité des solutions statiques et dynamiques est aussi soulignée.

L'apport de cette thèse se trouve aussi dans la conception et la réalisation d'une plateforme d'aide au placement. Une série d'algorithmes de complexité et de performances différentes ont été adaptés à ce problème précis. Soulignons l'utilisation d'un algorithme de recherche tabu qui n'avait pas été utilisé pour ce problème précis et qui a demandé une redéfinition de toutes ses composantes (liste tabu, critère d'aspiration, etc.).

Un des apports pratiques de ce travail réside dans le fait de livrer un produit complet et utilisable. En effet cela a demandé la réalisation de nombreuses interfaces telles que celles qui concernent les outils d'évaluation de performances, les outils de relevé de traces, le langage de description de graphes et de placement de la machine parallèle, l'analyse des tables de routage.

L'interfaçage de la boîte à outils de placement avec différents outils permettant de la valider et d'améliorer itérativement le placement constitue une phase originale et intéressante.

Le dernier apport de ce travail, et non le moindre, est d'avoir su tirer parti d'un ensemble de recherches existantes dans différents domaines et d'avoir fait une synthèse afin de réaliser un prototype. Ces domaines sont :

- L'évaluation de performances qui a permis de valider l'outil et d'améliorer celui-ci. L'interfaçage de l'outil de placement avec ANDES, un outil de génération de graphes synthétiques grâce aux collaborations entre l'équipe, dont je fais partie, de placement-ordonnancement dirigée par Denis Trystram du LMC-IMAG et l'équipe d'évaluation de performances dirigée par Brigitte Plateau du LGI-IMAG et plus particulièrement avec Joao-Paulo Kitajima est une phase importante du travail réalisé.

- La recherche en terme d'environnements de programmation, et plus particulièrement le travail de conception d'un nouvel environnement de programmation, comprenant un langage appelé Athapascan (travail réalisé au sein du projet APACHE, projet commun au LMC et LGI-IMAG)[Apa93].

- La recherche opérationnelle dont sont issus les algorithmes de placement tels que le tabu et le recuit simulé. Le tabu a été défini en collaboration avec l'équipe du professeur Jacek Błażewicz de l'Université Polytechnique de Poznan (Pologne) qui avait déjà, entre autres, appliqué un tel algorithme au problème de coupe à deux dimensions [BDSW91].

- La recherche en théorie des graphes, qui est à la base des algorithmes d'ordonnancement et plus précisemment l'algorithme de regroupement DSC de Tao Yang et Apostolos Gerasoulis avec lequel mes algorithmes de placement sont interfacés.

Ce travail fait aussi partie du projet européen Copernicus, sous-projet SEPP (Software Engineering for Parallel Processing). Le but de ce projet est de fournir un environnement de programmation complet destiné aux machines parallèles. Les sites de Barcelone, Grenoble et Bratislava sont chargés de founir une solution au problème du placement statique et dynamique[BCM+94].

Chapitre 2

Modèle de machines

2.1 Introduction

Dans ce chapitre, une rapide description des différents types de machines actuelles est fournie. Dans un premier temps, les caractéristiques des machines séquentielles sont rappelées et ensuite les machines parallèles à mémoire distribuée, qui sont similaires à un ensemble de machines séquentielles interconnectées, sont elles-aussi décrites. Ce chapitre montre la complexité des processeurs actuels qui fait que, entre autres, le coût d'exécution d'une instruction, ne comporte pas seulement le passage de celle-ci dans le processeur, mais aussi apporte d'autres répercutions sur le coût d'exécution du programme. Une série de paramètres utilisés dans la description des modèles de machines destinés aux problèmes de placement sont extraits de la masse des informations relatives aux machines. Le but à atteindre en spécifiant ces paramètres est de trouver un compromis entre recherche de performances et portabilité. Il s'agit aussi de ne pas énumérer l'ensemble des paramètres qui seront masqués par les différentes couches logicielles et par la granularité choisie qui est assez forte.

2.2 Machine séquentielle

Un ordinateur séquentiel tel que décrit par Von-Neuman à la fin des années quarante est composé d'une unité de traitement (processeur), d'une mémoire et de périphériques (disques, écrans, etc.). Cette description est encore à la base des ordinateurs séquentiels actuels. Elle peut cependant être raffinée.

2.2.1 Processeurs

Les processeurs actuels [Dow93] sont subdivisés en plusieurs unités qui résultent de la recherche de performances au moindre coût (cf figure 2.1). La notion

de performance conduit à dédoubler certaines des unités de traitement et à spécialiser celles-ci, tandis que la notion de coût de fabrication conduit à trouver des compromis tels que la mémoire cache.

Les processeurs actuels possèdent plusieurs unités spécialisées qui peuvent fonctionner en parallèle. En général on retrouve une unité destinée au traitement des nombres en virgule flottante et une unité destinée au traitement des entiers. De plus ces unités spécialisées peuvent elles-aussi être dédoublées, comme par exemple, dans les processeurs super-scalaires, plusieurs unités de traitement des scalaires cœxistent. La répartition des instructions parmi les différentes unités de traitement peut être réalisée à la compilation et/ou à l'exécution. Par exemple, dans les processeurs de type LIW[1] comme le i860, une instruction de base correspond à plusieurs traitements différents à effectuer par les différentes unités.

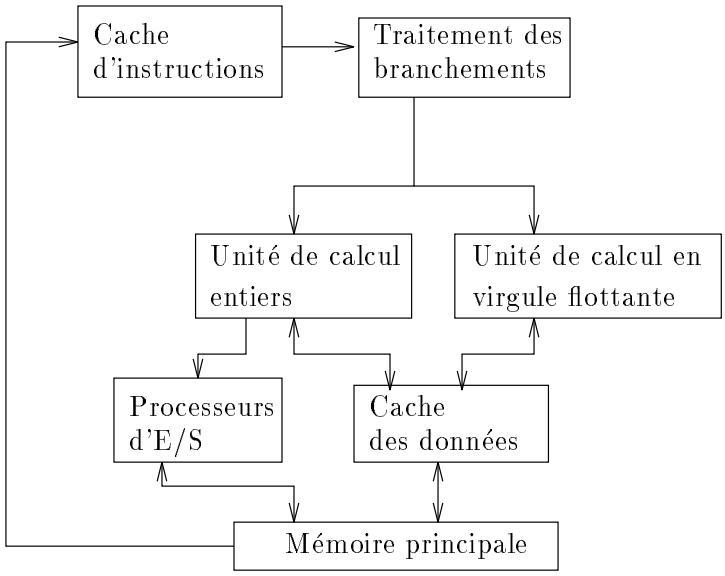


FIG. 2.1 - : Architecture du RS6000

Afin d'améliorer les performances, les constructeurs ont conçu des *pipelines* [Per87] [Dow93] à l'intérieur même des processeurs (cf figure 2.1). Les processeurs actuels peuvent comporter jusqu'à trois pipelines qui constituent donc des systèmes parallèles à l'intérieur même des processeurs. La notion de *pipeline* consiste à découper une unité fonctionnelle (traitement des instructions, traitement des références mémoires, opérations en arithmétique flottante) en plusieurs étages de manière à ce que plusieurs données à traiter puissent l'être en même temps. Une donnée à traiter va rentrer dans le premier étage, subir un premier traitement, passer au deuxième étage (pendant ce temps une autre donnée est rentrée dans le premier étage) et ce jusqu'au dernier étage. Il y a un temps de remplissage du

[1] LIW: *Long Instruction Word*

pipeline qui dépend du nombre d'étages, d. Si t est le temps de passage d'une donnée dans un étage, le temps de traitement de la première donnée sera donc $t * d$, tandis que la sortie des $n - 1$ données restantes prendra $(n - 1) * t$. Ce qui en théorie permettrait une amélioration de $\frac{n * d}{n + d - 1}$ par rapport au mode séquentiel. Cela en théorie seulement car les différents pipelines ne restent pas toujours occupés. Par exemple savoir quelles données fournir au pipeline d'instructions est problématique : en effet tant que le programme se déroule adresse par adresse, tout se passe bien ; mais dès que l'on rencontre une instruction de branchement (absolu ou pire, conditionnel), les instructions suivantes qui étaient déjà rentrées dans le pipeline peuvent ne plus servir à rien ... Afin d'éviter de tels désagréments des dispositions sont prises à la compilation (par exemple, avancer le branchement de plusieurs instructions, etc.) et lors de l'exécution (par exemple en choisissant de considérer qu'un branchement a lieu plus souvent en avant, on obtient expérimentalement de meilleurs résultats).

Les données à fournir à un pipeline doivent être assez semblables et découpables en morceaux de taille fixe de manière à permettre une fluidité de circulation des informations. Au niveau des instructions, les instructions simples (ne possédant pas de modes d'adressage trop complexes) sont les mieux adaptées. Cela, ajouté au fait que des études ont montré que, statistiquement parlant, seul un petit nombre d'instructions dans les processeurs CISC[2] étaient très utilisées, fait que la plupart des processeurs actuels possèdent des composantes de type RISC[3], c'est-à-dire, un jeu d'instructions à adressage réduit et des pipelines.

Les capacités de traitement des processeurs actuels sont de l'ordre de plusieurs centaines de millions d'instructions par seconde. Cependant leur complexité est telle que peu de compilateurs et d'applications parviennent à en tirer parti. Un des seuls moyens d'utiliser de manière efficace ces processeurs est d'écrire les routines les plus utilisées directement en langage d'assemblage (par exemple les routines de base utilisées en algèbre linéaire, les BLAS[4]). Mais avec un compilateur classique, le nombre moyen d'instructions traité par seconde tombe généralement à moins de la moitié des capacités de pointe du processeur.

2.2.2 Mémoire

Le prix actuel des mémoires à accès rapide forcent les constructeurs à en utiliser aussi peu que possible. La solution apportée à ce problème est d'utiliser des mémoires caches rapides qui contiennent les données les plus utilisées et font fonction de tampon entre le processeur et la mémoire principale (plus lente et moins chère). La gestion de telles mémoires consiste à charger le cache de manière à minimiser le nombre d'accès à la mémoire principale. Ce principe se retrouve

[2]CISC: Complex Instruction Set Computer
[3]RISC: Reduce Instruction Set Computer
[4]BLAS: Basic Linear Algebra Sub-routines

à d'autres niveaux et on peut décrire les différents types de stockage disponibles sur les machines sous forme d'une pyramide (cf schéma 2.2). Au sommet de celle-ci, on retrouve les plus chères, les plus performantes mais aussi les plus petites mémoires ; au plus on descend dans la pyramide, au plus les performances et le prix au kilo-octets sont moindre et les capacités de stockage augmentent.

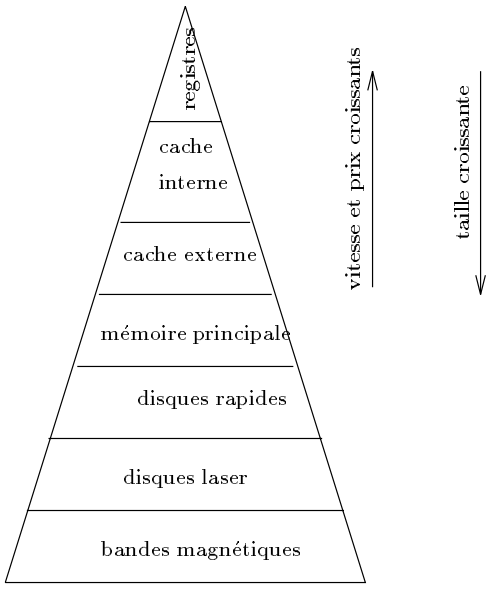


FIG. 2.2 - : Hiérarchie des mémoires

Classiquement, comme par exemple dans le RS6000 (cf figure 2.1), le flux d'instructions qui provient de la mémoire sera dirigé vers un cache et ensuite suivant leurs types (arguments flottants ou entiers) vers plusieurs unités de traitement. Il existe aussi des mémoires cache pour les données. La taille mémoire des stations de travail actuelles est de l'ordre de 64 Méga-octets et leur temps d'accès est 60 nano-secondes.

Les périphériques comportent des disques à accès rapides (un disque moyen actuel permet de stocker deux giga-octets, transfère 6 méga-octets par seconde et a un temps d'accès moyen de 10 milli-secondes), ainsi que des périphériques d'archivage tels que les DAT[5], CD-réinscriptibles et cassettes vidéo 8mm (un DAT actuel permet de stocker 8 Giga-octets et un disque laser, 600 méga-octets).

2.2.3 Les réseaux

Les besoins croissants d'échanges d'information de tous genres ont conduit ces dernières années à un développement phénoménal des moyens de communication. Ces besoins s'expliquent du fait du caractère central de l'information dans la prise

[5]DAT: Digital Audio Tape

de décision et aussi du fait de la mondialisation des échanges. Le besoin de performances dans ce domaine est critique. Du point de vue des réseaux informatiques [Tan88], nous avons vu se développer ces dernières années des réseaux locaux de plus en plus performants (Ethernet (20Mbits/s), FDDI (100Mbits/s), etc.), mais aussi ces différents réseaux s'interconnecter afin de constituer les différents réseaux mondiaux qui eux-même sont interconnectés (Internet, Bitnet, Compuserve, etc.). Internet, le plus grand réseau, qui de base reliait des centres publics (université, grands centres de recherche et armée) s'ouvre de plus en plus au privé et même au grand public. De plus la technologie ATM (600 Mbits/s), qui commence à être utilisée ne restreint plus la fonctionnalité des réseaux (Multimédia). C'est-à-dire que sur le même réseau passeront non seulement des informations destinées aux ordinateurs mais aussi des images (télévision) et des sons (qualité numérique).

2.3 Machine parallèle à mémoire distribuée

Dans la présente thèse, seules les machines parallèles à mémoire distribuée [HB93] sont traitées. Remarquons que nombre de machines récentes sont basées sur ce concept (SP1 d'IBM, CM5 de TMC, T3D de Cray, Paragon d'Intel, CS2 de PCI, etc.) qui paraît le plus prometteur. Ces machines sont constituées d'un ensemble de nœuds communiquant via un réseau d'interconnexion. Chaque nœud est semblable à un ordinateur séquentiel, il possède des capacités de traitement de l'information (processeur) et des capacités de stockage (mémoire).

2.3.1 Les unités de traitement

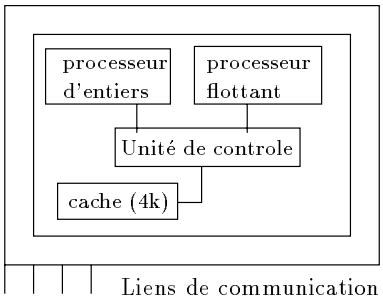


FIG. 2.3 - : Schéma synoptique du transputer T800

Certains constructeurs (Inmos, Ncube) ont développé des processeurs spécialement dédié au calcul parallèle de par leurs capacités de communication. Le transputer T800(cf figure 2.3), par exemple, dispose de quatre liens d'entrées-sorties qui permettent de communiquer tout en continuant à utiliser l'unité de

traitement d'entiers et de flottants. Un transputer T800 comprend une unité de calcul entier (10 MIPS[6]), une unité flottante (2 MFLOPS[7]) et quatre liens d'entrées-sorties (10 Mbits/sec) qui peuvent fonctionner en parallèle. Cependant la production de masse des processeurs classiques a permis à ces derniers d'être très performants et très peu onéreux, ce qui a amené de nombreux constructeurs à choisir d'utiliser de tels processeurs comme éléments de traitement de leurs machines parallèles. Pour pallier leur manque d'aptitudes à la communication, certains ont décidé de doubler le nombre de processeurs par nœud (un pour le calcul et un pour les communications). Par exemple, IBM a choisi le Power2 comme processeur de calcul du SP2, et l'Intel i860 comme processeur de communication.

2.3.2 Le réseau d'interconnexion

Les processeurs sont connectés via un réseau rapide. Deux grandes familles de réseaux peuvent être distinguées : les réseaux point-à-point où les processeurs de calcul constituent aussi les nœuds du réseau de communication (hypercube de transputers, Ncube, etc) et les réseaux multi-étages où les processeurs de calcul ne s'occupent que des communications qui les concernent directement (SP1, CS2, etc).

Dans la première famille (cf figure 2.4), le routage d'informations qui ne leur sont pas destinées, empiète sur le temps de calcul des processeurs, tandis que dans la deuxième, ce routage est effectué automatiquement par les nœuds intermédiaires du réseau et n'interfère donc pas sur le temps de calcul des autres processeurs. Notons néanmoins qu'un réseau chargé, quel qu'il soit interfére sur les communications qui doivent avoir lieu (le coût d'une communication dépendra de la charge du réseau [TP94]).

Les réseaux point-à-point

Les réseaux point-à-point sont facilement formés à l'aide des processeurs dédiés au parallélisme et possédant une série de liens de communications. Il suffit de les relier afin de former différentes topologies. Les contraintes technologiques et économiques font que le nombre de liens d'interconnexion de chaque processeur est faible : de 4 pour les transputers d'Inmos à 13 pour le Ncube2 de Ncube. Les topologies sont choisies dans le but de minimiser le diamètre, c'est-à-dire la distance maximale à parcourir entre deux nœuds, et la distance moyenne. Parmi ceux-ci, les réseaux de transputers forment les réseaux point-à-point parmi les plus courants en Europe. Ce type de réseaux consiste à connecter directement les liens de communication des transputers entre eux (par exemple en tore, cf 2.4).

Une autre technologie, appelée machines reconfigurables, est très utilisée dans

[6] 1 MIPS=unité de mesure valant 1 million d'instructions entière par seconde

[7] 1 MFLOPS=unité de mesure valant 1 million d'instructions en virgule flottante par seconde

le cadre des réseaux de transputers. Elle consiste à placer des *crossbar switches*[8] entre les différents processeurs. Le coût temporel assez élevé de configuration de tels *switches* est tel que généralement, dans une première phase un réseau fixe est généré et ensuite dans une deuxième phase, le programme est exécuté sur celui-ci. La communication entre deux nœuds qui ne sont pas directement connectés demande des mécanismes de routage logiciel.

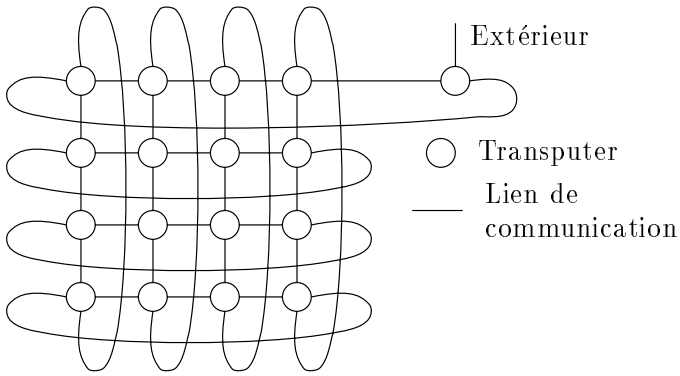


FIG. 2.4 - : Tore tronqué constitué de transputers

Les réseaux multi-étages

Dans les réseaux multi-étages, les nœuds constituant le réseau diffèrent des unités de traitement. Les temps de latence des réseaux actuels sont très peu élevés. Certains réseaux, tels que le *switch* rapide équipant la SP2 d'IBM forcent chaque communication à passer par un nombre fixe de liens pour une taille de machine donnée. D'autres tels que le *fat-tree* de la CM5 de TMC font que la durée des communications dépend de la distance dans le réseau. Remarquons que cette différence est minime et négligeable par rapport aux temps de latence logicielle qui ne dépendent pas de la distance entre processeurs.

[8] *crossbar switch :* boitier comportant un nombre identique d'entrées et de sorties et de relier n'importe laquelle de ces entrées avec n'importe laquelle des sorties

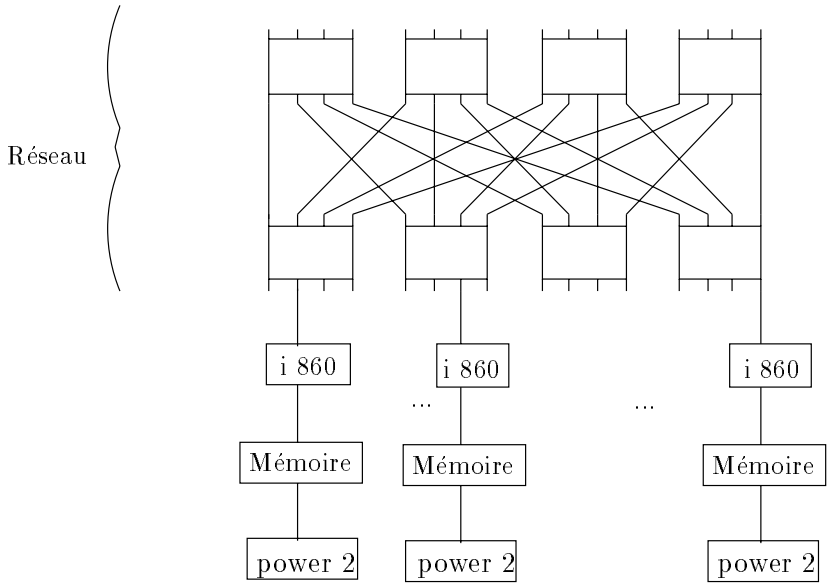


FIG. 2.5 - : SP2 d'IBM avec 16 nœuds

La principale différence avec un réseau local (Ethernet, FDDI, etc.) réside dans le fait que l'ensemble des processeurs est regroupé physiquement à un même endroit, ce qui permet de diminuer le taux de panne, appelé aussi MTBF (*Mean Time Between Failure*), de l'ensemble du système et permet de diminuer les préoccupations en termes de tolérance aux fautes. La localisation précise d'une machine parallèle diminue aussi les problèmes de sécurité en facilitant le contrôle des accès à la machine et permet de réserver plus facilement une partie des nœuds afin de satisfaire une application qui aurait de grands besoins en puissance de calcul. Les premières machines parallèles étaient mono-utilisateurs mais avec l'apparition de systèmes d'exploitation complets sur les différents nœuds cette caractéristique a tendance à disparaître.

2.3.3 Connexion vers l'extérieur

Généralement les machines parallèles sont connectées au réseau local et souvent via une machine appelée frontale ou hôte. Cette machine frontale permet d'ouvrir la machine parallèle au monde extérieur : périphériques de visualisation, de stockage de masse, etc. Sur cette machine peuvent tourner une série d'outils tels que les compilateurs et autres outils d'aide au développement (débogueur, visualisation de traces, etc). Son utilisation permet aussi de pallier au manque de logiciels tournant sur machines parallèles.

2.4 Paramètres matériels

Vue la complexité des machines existantes, il est difficile de prendre tous les paramètres en compte dans les modèles de machines. De plus, une grande partie des caractéristiques matérielles sont masquées par les différentes couches logicielles. Une série de paramètres peuvent être dégagés afin de définir les modèles de machines trouvés dans la littérature.

- **Le nombre de processeurs**
  Dans les différents modèles de machines utilisés, le nombre de processeurs est considéré comme fixé ou non fixé. Dans le deuxième cas, il faudra souvent effectuer une projection de ce réseau virtuel vers le réseau réel.

- **Performance de calcul**
  Trois modèles de processeurs sont généralement utilisés :

  Processeurs homogènes
  Les processeurs sont tous identiques tant en ce qui concerne leurs possibilités que leur vitesse. Ceci est le cas sur bon nombre de machines parallèles telles que les réseaux de transputers, la SP1, la CM5, la CS2, la Paragon, etc.

  Processeurs uniformes
  Les processeurs ont les mêmes possibilités mais ont des vitesses de traitement différentes. Ce qui peut être, par exemple, le cas d'un réseau de Sparcs possédant des processeurs Sparcs de plusieurs générations.

  Processeurs hétérogènes
  La vitesse de traitement dépend du type d'opération et aussi du processeur. Ce qui est le cas de la plupart des réseaux Ethernet, sur lesquels on retrouve aussi bien des IBM, des HPs, des Crays, etc.

- **Processeurs dédiés**
  Sur certaines machines, tous les nœuds ne peuvent pas effectuer toutes les opérations. Par exemple, seul le nœud relié à l'extérieur sur le Méganode, réseau de transputers de Telmat, peut effectuer des lectures/écritures sur fichiers (ces fichiers sont situés sur une machine hôte).

- **Le recouvrement calcul/communication**
  Certaines machines possédant des processeurs spécialisés permettent d'effectuer les communications en même temps que le calcul. Notons que dans le cadre du placement, les boîtes noires constituant les tâches nous empêchent de distinguer les communications qui seront recouvertes de celles qui ne le seront pas.

- **La notion de voisinage**
  Certaines machines sont complètement connectées comme les réseaux multi-étages et le coût d'une communication sur une telle machine ne dépend pas du processeur émetteur ni du processeur récepteur. D'autres machines, au contraire, dont la plupart des machines point-à-point ne sont pas complètement connectées. Sur ces machines, un processeur donné ne peut communiquer directement qu'avec un nombre donné de processeurs qui lui sont voisins. Le coût d'une communication va donc dépendre de la distance, c'est-à-dire du nombre de nœuds intermédiaires nécessaires à la communication.

- **1-port/δ-port**
  Certaines machines ont la possibilité d'utiliser en même temps leurs différents liens de sortie (δ-ports de communication), d'autres non (1-port).

- **Paramètres supplémentaires**
  D'autres types de paramètres peuvent être pris en compte, tel que la gestion de ressources mémoire, etc. Par exemple de nombreuses machines types réseaux de transputers ne possèdent qu'un méga-octet par nœud, ce qui limite fortement le type de programmes (et de données) qui peut y être exécuté.

2.5 Objectif

Différents paramètres ont été tirés des caractéristiques précédemment décrites afin de modéliser certains aspects de ces machines. Le but de ces modélisations sera de :

- rester suffisamment proche de la machine afin de pouvoir profiter un maximum des performances de celle-ci,
- être suffisamment éloigné de la machine afin de pouvoir représenter un ensemble suffisamment large de machines de manière à ce que les développements futurs soient portables,
- dégager les composantes principales en ne tenant pas compte de trop de détails techniques qui ne feraient que gêner les raisonnements,
- essayer d'anticiper les tendances générales du marché. C'est-à-dire faire des recherches sur les machines de demain et non du passé.

Par la suite nous verrons parmi les différents modèles utilisés :

- Des modèles utilisés le plus souvent dans le cadre de l'ordonnancement qui bien que proches d'autres types de machines telles que les machines SIMD

ou les machines MIMD à mémoire partagée, sont en général assez éloignés des machines MIMD à mémoire distribuée. Parmi ceux-ci, on retrouve :

  - Des machines homogènes avec m processeurs et avec coût de communication nul [CG72].
  - Des machines homogènes avec m processeurs et avec coût de communication unitaire [RS87].
  - Un nombre limité de machines (2 ou 3) uniformes avec de fortes restrictions sur les communications [BBGT93].
  - Des architectures idéalisées, c'est-à-dire le plus souvent un nombre de processeurs homogènes à discrétion.

- Des modèles plus proches de la réalité des machines MIMD à mémoire distribuée, tels que :

  - Un modèle considérant un réseau de processeurs homogènes interconnectés par un réseau multi-étage qui nous servira de base pour l'outil de placement.
  - Des modèles considérant un réseau de processeurs homogènes connectés point-à-point [Bok81].

2.6 Conclusion

Dans ce chapitre, nous avons donc constaté qu'une machine parallèle MIMD à mémoire distribuée est en fait une collection de machines séquentielles interconnectées par un réseau. Le modèle qui nous semble le plus prometteur, à court et moyen terme, est celui proposé par de nombreux constructeurs aujourd'hui : une série de processeurs homogènes connecté via un réseau multi-étages avec un coût des communications inter-processeurs ne dépendant pas de ceux-ci (tous les processeurs sont à distance constante ou sont considérés comme tels). C'est ce modèle qui est la base de tous nos choix futurs. Parmi les constructeurs qui ont fait ce choix on retrouve IBM avec le SP2, PCI avec la CS2 et TMC avec la CM5. Sur les autres machines actuelles telles que la Paragon d'Intel, le T3D de Cray qui ont encore un réseau point-à-point les performances de ceux-ci (processeurs de routage dédiés, etc) alliées au coût de latences logicielles, diminuent complètement l'importance de la distance entre processeurs.

Cependant pour des raisons historiques, les tests effectués par la suite le seront sur réseaux de transputers et nous essayerons d'adapter nos algorithmes afin de tenir compte de la distance entre processeurs. Pour réaliser ceci, des outils d'analyse des tables de routages seront développés, ainsi que des tests permettant de déterminer de manière exacte les temps de routage pour une topologie donnée (le tore).

Chapitre 3

Modèle de programmation

3.1 Introduction

Dans ce chapitre, dédié aux composantes logicielles, les systèmes d'exploitation des machines parallèles actuelles sont décrits et les différents paramètres représentants les possibilités des systèmes sont soulignées. Ces paramètres vont permettre de définir une série de modèles de description de programmes parallèles. Parmi ceux-ci, deux modèles sont décrit plus particulièrement : les graphes acycliques de précédence (DAG) et les graphes de tâches sans précédence.

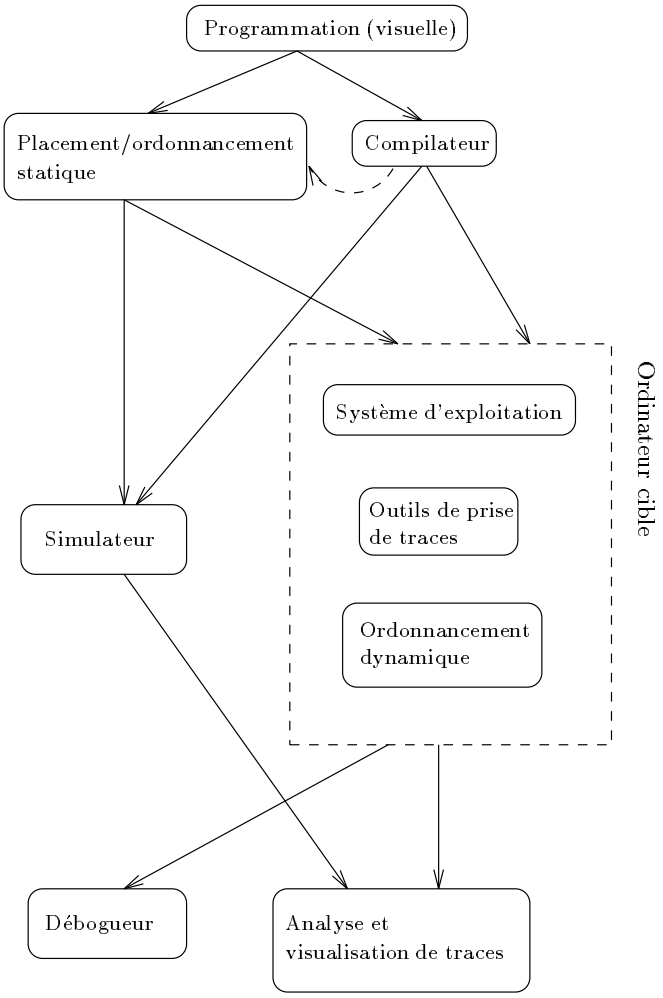


FIG. 3.1 - : Environnement de programmation parallèle

Le schéma 3.1 représente un environnement de programmation parallèle complet tel que défini dans le cadre du projet SEPP. Remarquons que de nombreux arcs pourraient le compléter, tel qu'un arc allant du simulateur au débogueur, etc.

3.2 Environnement

Autour d'un ordinateur parallèle, peut venir s'articuler un grand nombre d'outils (cf figure 3.1)[HB93][Tan92]. Parmi ceux-ci, on trouve des langages de haut niveau (par exemple, graphique ou Prolog) et leurs compilateurs. On y trouve aussi les outils de placement/ordonnancement statique permettant de générer automatiquement une description de l'allocation des parties de codes sur les dif-

férents processeurs. Une série de mécanismes peuvent être mis en œuvre lors de l'exécution d'un programme sur une machine cible :

- un système d'exploitation qui en plus des bibliothèques standard contient des fonctions de gestion du parallélisme,
- des outils de prise de traces qui permettent grâce à une étude du comportement du programme d'améliorer de manière dynamique le placement et permettent aussi de fournir des indications au débogueur,
- Un outil de placement dynamique qui permet de répartir dynamiquement la charge de calcul et de communication et donc de la réguler.

La vie d'un programme parallèle dans un environnement complet comporte les étapes suivantes :

1. Conception et écriture à l'aide d'un langage de programmation. Diverses annotations relatives au parallélisme, par exemple concernant le graphe de tâches, peuvent être rajoutées au texte du programme afin de permettre une exploitation plus aisée de celui-ci (par exemple, l'extraction des données du graphe de tâches).
2. Compilation générant des codes objets du programme, mais aussi une description des différentes composantes du programme parallèle.
3. Placement/ordonnancement statique des composantes sur l'architecture cible.
4. Utilisation d'un simulateur de machine parallèle. Cette phase a l'intérêt d'être moins coûteuse qu'une vraie exécution sur la machine parallèle, mais a le défaut qu'il s'agit d'un modèle simplifiant la machine cible. Cette phase peut amener à améliorer le programme et donc nous ramener à la première étape.
5. Exécution du programme sur la machine cible. Durant cette exécution des outils de débogage interactifs peuvent être utilisés.
6. Après l'exécution du programme, les différentes informations enregistrées lors de celle-ci permettent :
   - de déboguer le programme de manière post-mortem. La prise de traces devrait permettre de réexécuter le programme de manière déterministe de façon à pouvoir reproduire les erreurs [FdK94] et à les corriger plus facilement,
   - de visualiser les performances du programme afin d'améliorer celui-ci,
   - de comparer les performances de différents programmes afin de choisir le plus indiqué,

– d'améliorer le placement statique effectué.

3.3 Système d'exploitation

3.3.1 Services offerts sur les nœuds

Sur les premières machines parallèles (réseau de transputers), les systèmes d'exploitation n'étaient le plus souvent présents que sous formes de bibliothèques de gestion des communications, des processus, etc. Par la suite sont apparus divers modules qui forment un début de couche système tels que des outils de prise de traces, des routeurs, etc. Actuellement, des systèmes Unix complets ou allégés se retrouvent sur chacun des nœuds (SP1, CS2, etc.). Ceci est le résultat de trois phénomènes :

- Les nœuds des machines parallèles à mémoire distribuée actuelles sont de véritables stations de travail, possédant un grand espace mémoire (64MO sur la SP1), une grande puissance de calcul (plus de 100 mips sur la SP1) et même parfois des possibilités de stockage de masse propre (minimum 1 GO de disque sur la SP2).

- Un compromis a dû être trouvé entre le fait de tirer parti à tout prix des performances des machines parallèles et l'objectif de programmabilité de ces machines.

- Un objectif de portabilité est aussi apparu. En effet les générations de machines se succèdent rapidement et la réécriture d'un code est coûteuse.

Les différentes possibilités des systèmes d'exploitation classiques existent aussi pour les systèmes d'exploitation des machines parallèles [Tan92] :

- **Pseudo-parallélisme**
  La notion de pseudo-parallélisme permet d'allouer différents processus à un même processeur et de faire exécuter de manière alternative des parties de ces processus, ce qui donne à l'utilisateur l'impression de parallélisme. Il existe différents types de pseudo-parallélisme :

  systèmes non-préemptifs
  Chaque processus qui a la main peut la rendre afin de permettre au processeur de continuer à exécuter le suivant. On retrouve ce mode dans des systèmes tels que MS-Windows de Microsoft.

  systèmes semi-préemptifs
  Un processus ne peut passer la main (être *déschédulé*) que sur certaines

instructions (sauts, communications, etc.). Un certain intervalle de temps est alloué à chaque processus et si un processus n'a pas rendu la main à la fin de son intervalle, il la rend automatiquement à la prochaine instruction *déschédulante* rencontrée. Ce système est utilisé par les transputers T800 d'Inmos.

systèmes préemptifs
Un certain intervalle de temps est alloué à chaque processus et à la fin de cet intervalle le système donne la main au processus suivant. Ce système se retrouve sur bon nombre de systèmes d'exploitation actuels tels que Unix.

Chaque processus possède un contexte propre, constitué par son espace mémoire, y compris la pile, et par les valeurs des différents registres du processeur. Cette possibilité est généralement disponible sur la plupart des machines actuelles.

Une file d'attente des processus est donc le plus souvent gérée par le système. La procédure généralement employée est de donner la main au premier processus rentré et qui est prêt. Ce mécanisme peut être plus complexe quand le système permet de donner des priorités différentes aux processus ; dans le cas du T800, par exemple, deux types de priorité existent (haute et basse) et correspondent à deux files d'attente différentes. La caractéristique des processus de priorité haute est de ne pas être *déschédulé* automatiquement au bout d'un certain laps de temps.

- **Processus légers**
  La notion de processus légers consiste à considérer que plusieurs unités d'exécutions peuvent être lancées en pseudo-parallélisme à moindre coût de mise en route et partager une zone de mémoire commune. Cette fonction n'est pas souvent fournie de base sur les systèmes actuels.

- **Mémoire paginée**
  La notion de mémoire paginée permet à un nœud couplé à un disque de gérer des tâches dont la taille mémoire dépasse la taille de la mémoire vive du nœud, en chargeant et déchargeant à la demande du système celle-ci sur le disque.

3.3.2 Services offerts entre les nœuds

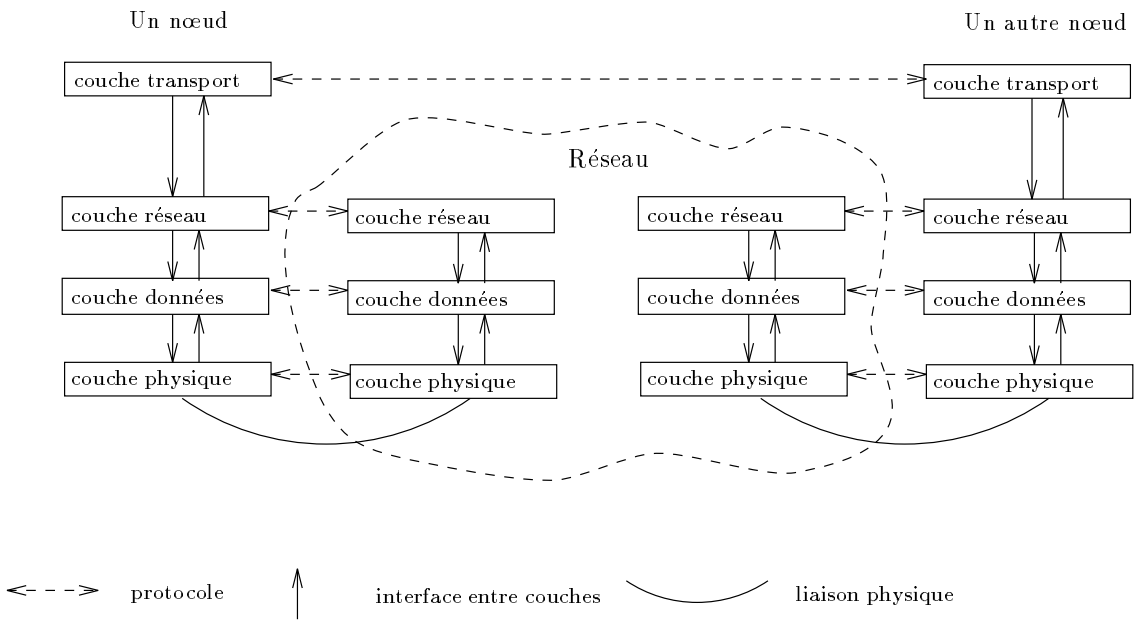


FIG. 3.2 - : Découpe en couches du système de communications

Les possibilités de communications offertes par un système peuvent être plus ou moins développées (cf figure 3.2)[Tan88]. L'Organisation des Standards Internationaux (OSI) a fourni une découpe fonctionnelle des possibilités de communications dans les réseaux. Les protocoles des trois premières couches forment le standard X25, dans la terminologie TCP/IP[1]. Ce qui caractérise cette partition des services, est d'une part la notion de protocole (description de la discussion permise entre deux couches d'un même niveau sur deux machines différentes) et d'autre part la notion d'interface (description des échanges d'une couche avec la couche supérieure et la couche inférieure sur la même machine). Les différentes couches composant X25 sont :

- La première couche, couche physique, est destinée à permettre le transfert binaire d'informations d'un nœud vers un autre. Elle correspond à un moyen physique minimal de communications avec l'extérieur.
- La deuxième couche, couche liens de données, permet d'acheminer, d'un nœud à son voisin direct, un message (ensemble d'octets).
- La troisième couche, couche réseau, a connaissance du fait que le nœud fait partie d'un réseau et gère des tables de routage (si besoin en est) de ce

[1]TCP/IP: Transmission Control Protocol/Internet Protocol

réseau. Les services proposés aux couches supérieures permettent de transférer des messages d'un point à un autre du réseau.

Les couches supérieures sont là pour permettre des contrôles de flux, le codage des données, la notion de session, etc.

Les différentes couches peuvent soit être implémentées à l'aide de dispositifs matériels, soit à l'aide de composantes logicielles. Par exemple, dans les réseaux de transputers, les deux premières couches se retrouvent directement au niveau du processeur, tandis que la troisième est réalisée à l'aide de routeurs logiciels tels que VCR[2], ou celui contenu dans le langage C d'Inmos. Dans le SP1, les trois premières couches sont réalisées principalement de manière matérielle grâce au *switch* rapide qui permet de considérer le réseau comme complètement connecté.

Certaines machines telle la CM5 permettent les communications globales de manière matérielle.

3.3.3 Utilisation d'un processeur

Un processeur peut se trouver dans différents états :

Inactif

Deux types d'inactivité sont envisageables.

- Attente d'une communication - Tous les processus situés sur le processeur sont en attente de communication (l'émetteur ou le récepteur avec lequel ils désirent entrer en communication n'est pas prêt).
- Fin de programme - Tous les processus situés sur le processeur ont terminé leur exécution.

Actif

Trois types d'activité sont envisageables.

- Communication - Le processeur est responsable en tout ou en partie d'une ou plusieurs communications qui ont lieu.
- Travail - Le travail actuellement effectué fait partie du code d'un des processus utilisateurs.
- Système - Le travail actuellement effectué fait partie des différentes charges système du processeur (gestion du pseudo-parallélisme, outils de monitoring, gestion de la pagination, etc.).

[2] VCR : *Virtual Channel Router* de l'université de Southampton

3.4 Modèles de programmes

De nombreux modèles de programmes ont été proposés dans la littérature. Parmi ceux-ci, deux des plus utilisés sont les graphes acycliques de précédences, appelés aussi DAG[3]) et les graphes de tâches sans précédence.

3.4.1 Paramètres logiciels

Voici quatre possibilités des systèmes d'exploitation que l'on retrouve dans la littérature sur l'ordonnancement.

- **Routage ou non**
  Dans les réseaux autres que complètement connectés (clique), la possibilité d'utiliser un système de routage permet de placer des tâches communicantes sur des processeurs non-voisins. Dans ce cas, le coût de routage devra être pris en compte. Dans le cas d'un réseau de processeurs non totalement connecté et sans routeur, les tâches communicantes devront être placées sur un même processeur ou sur des processeurs voisins. Remarquons que le cas où il n'existe pas de mécanisme de routage fait de plus en plus partie d'une époque révolue.

- **Préemption**
  La préemption permet à une tâche d'être momentanément interrompue lors de son exécution afin de laisser le processeur à une autre tâche. Ce mécanisme est généralisé lors de l'utilisation de la gestion de pseudo-parallélisme du système.

- **Duplication**
  La duplication de tâches permet, entre autres, d'éliminer une série de communications inter-processeurs en dupliquant le code des tâches[Chr89]. Ce mécanisme est encore peu usité de manière fine : actuellement, soit on ne duplique pas, soit on duplique tout et partout (programmation de type *SPMD*).

- **Migration**
  La migration de tâches permet une répartition dynamique de la charge. Dès lors, celle-ci n'est pas utilisée dans la littérature dans le cadre statique. Dans le cadre dynamique, fort coûteuse, elle est souvent remplacée par de l'activation à distance.

[3]DAG: Direct Acyclic Graph

3.4.2 Graphe de précédence

Définition : *Un graphe de précédence* est un graphe acyclique dont les sommets représentent des tâches qui sont caractérisées par un coût d'exécution et dont les arcs représentent des relations de précédence entre tâches et peuvent être caractérisés par des coûts de communication [Sar89] [**?**].

La figure 3.3 montre une telle représentation pour le calcul d'une expression arithmétique simple.Dans ce modèle, une tâche ne s'exécute que quand tous ses prédécesseurs ont terminé de s'exécuter, reçoit le cas échéant des données de ceux-ci, exécute les traitements et fournit le cas échéant en sortie des données à ses successeurs.

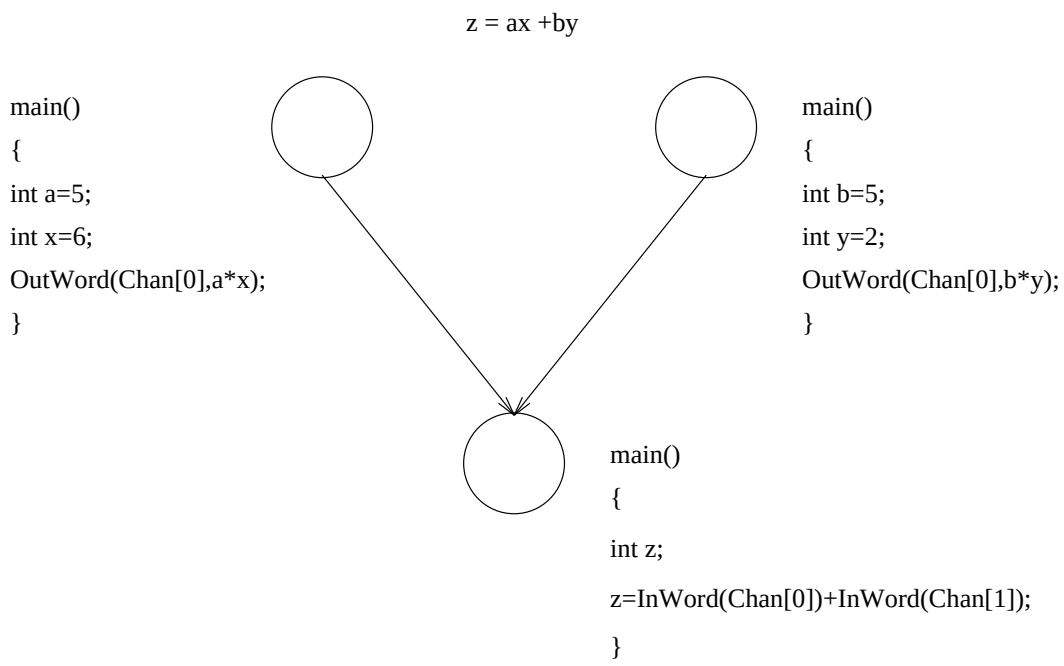


FIG. 3.3 - : Exemple de graphe de précédence pour le programme de calcul de $z = ax + by$

Définition : Le problème de l'*ordonnancement*, consiste à affecter à chacune des tâches d'un graphe de précédence un processeur et une date de début d'exécution.

Différentes adaptations et relaxations de la représentation sous forme de graphe de précédence ont été étudiées dans la littérature. La plupart considèrent le réseau de processeurs comme complètement connecté et disposent soit d'un nombre fixe de processeurs soit d'autant de processeurs (virtuels) qu'il est nécessaire, dans

ce cas on parle alors plutôt de *regroupement*. Notons qu'il existe aussi d'autres modèles qui possèdent des tâches multi-processeurs [BESW93].

Les algorithmes réguliers tels que ceux de l'algèbre linéaire s'expriment naturellement sous forme de graphe de précédence aboutissant à une implémentation efficace sur réseau de processeurs. Cette démarche tend à dicter une méthodologie de programmation visant à construire des graphes de précédence. La littérature est de plus en plus abondante sur la résolution des problèmes de placement vus du côté de l'ordonnancement, c'est-à-dire en prenant en compte des dépendances et des communications. Ces travaux sont souvent théoriques (architecture idéalisée, temps d'exécution constants des tâches, communications en temps constant, etc.) et peu de réalisations pratiques sont proposées [ERL90].

3.4.3 Graphes de tâches sans précédence

```
#define nbre_travaux 23
#define nbre_esclave 3
extern int donne_travail();

main() /* source du maître */
{
int s,busy,esclave=0;
int nt=nbre_travaux;
while (esclave++<nbre_esclave)
    if (nt!=0) {OutWord(Chan[esclave],donne_travail());nt--;busy++;};
while (busy)
{
esclave=AltBlck(&Chan[0],nbre_esclave);
s+=InWord(Chan[esclave]);busy--;
if (nt)
    { OutWord(Chan[esclave],donne_travail());nt--;busy++;};
};
}


extern int travail(int t);
main() /* source des esclaves */
{
int t;
while (1==1)
{
t=InWord(Chan[0]);
OutWord(Chan[0],travail(t));
};
}
```

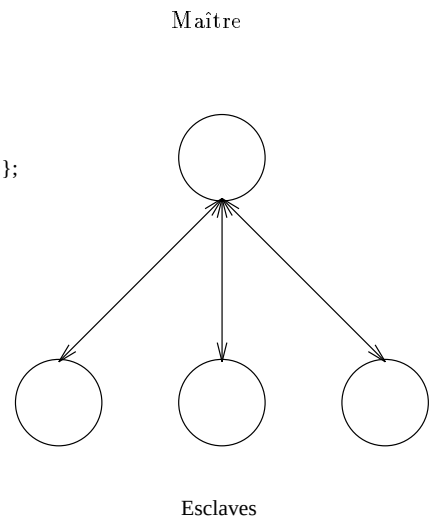


FIG. 3.4 - : Exemple de graphe de tâches

Il existe une autre manière de modéliser un algorithme sans relations de précédence entre les tâches, une arête modélisant alors simplement un coût total de communication sans se soucier de leurs sens[NT93][Sto77].

Une tâche dans ce modèle est une boîte noire qui peut calculer et communiquer alternativement (ou en même temps suivant la machine), les seules indications disponibles étant les coûts totaux de calcul des tâches (qui vont pondérer les nœuds du graphe) et les coûts totaux de communication de celles-ci (qui vont pondérer les arêtes).

Définition : Le problème du *placement* consiste à attribuer à chacune des tâches un processeur sur lequel elle va s'exécuter. Les différentes unités d'exécution constituant la tâche vont être le plus souvent gérées par le mécanisme de gestion de pseudo-parallélisme du système d'exploitation.

Définition : La notion de *tâche* dans la littérature et dans la suite de cette thèse, dépend donc du contexte utilisé : boîte noire caractérisée par un coût moyen de calcul et de communication dans le cadre du placement et unité d'exécution ayant une série de communications en entrée et en sortie dans le cadre de l'ordonnancement.

La figure 3.4 montre l'exemple d'une telle représentation pour un calcul maître-esclave. Nous appellerons cette représentation *graphe de tâches sans précédence*. C'est ce modèle qui sert de base à la plupart des algorithmes de placement.

Ce modèle peut servir à représenter plusieurs réalités. Les spécialistes de réseaux y verront un modèle client-serveur où les coûts de calculs représentent un coût moyen d'utilisation des serveurs tandis que le coût de communications représentera le coût moyen d'appel des services des serveurs.

Les spécialistes des graphes de précédence peuvent y voir une relaxation de leur modèle avec préemption. Cette vue à mon avis restreint fortement les capacités de représentation de ce modèle.

Les spécialistes de compilation considéreront la notion de tâche comme une unité d'allocation mémoire. C'est ce dernier paradigme qui a conduit à l'utilisation massive de ce modèle. En effet dans la plupart des langages MIMD classiques, l'utilisateur doit d'une part fournir le code des différentes tâches, mais aussi fournir le placement de ces tâches sur les différents processeurs afin de permettre le chargement du programme et souvent (Inmos C, C3L, etc.) une description des communications entre tâches afin de fournir ces indications aux mécanismes de routages. Cette obligation, comprend donc la description du graphe de tâches. Il suffit de demander à l'utilisateur des estimations des pondérations afin d'obtenir le modèle complet.

L'utilisation du terme *tâche* dépend donc du contexte où il est utilisé.

3.4.4 Autres modèles

Les deux modèles vus précédemment restreignent les possibilités de programmation des machines MIMD. En effet vouloir modéliser tout programme avant exécution demande d'avoir une connaissance précise du comportement à l'exécution de ce programme. Cette connaissance n'est pas toujours disponible avant l'exécution, que ce soit parce que celle-ci dépend fortement des données fournies au programme ou que ce soit dû à la difficulté d'extraction de cette information (de l'esprit du programmeur ou du code source du programme).

Parmi les possibilités qui ne sont pas représentables par les graphes de tâches ou par les DAGs, on trouve la création dynamique de tâches et la migration de celles-ci[BGT94]. Des outils de répartition dynamique de charge peuvent être utilisés pour gérer cela. Les deux vues, statique et dynamique, ne semblent pas opposées. En effet pourquoi ne pas utiliser au maximum les informations disponibles à la compilation et ensuite affiner celles-ci lors de l'exécution. Répartir les différents codes sur le réseau avant de lancer l'exécution diminue le temps d'exécution en permettant un démarrage parallèle de l'application plus efficace. Devant la diversité des méthodes possibles d'allocation et régulation de la charge, Casavant et Kuhl ont proposé une classification très complète des différentes méthodes dans le cadre de machines parallèles à mémoire distribuée [CK88]. Ces méthodes peuvent être envisagées comme la description des interactions entre des ressources, des consommateurs et une politique d'attribution de ces ressources aux consommateurs. Les buts recherchés sont principalement l'*efficacité* et la *performance*. Dans le premier cas, le but est de satisfaire les consommateurs en utilisant peu de ressources, dans le second, il s'agit plutôt d'obtenir le maximum de satisfaction. Ces deux critères sont (évidemment) contradictoires. La classification proposée distingue principalement cinq caractéristiques :

- stratégies statiques ou dynamiques, selon que l'allocation des tâches s'effectue une fois pour toutes avant l'exécution ou est régulée pendant l'exécution,
- équilibrage de la charge,
- stratégies déterministes ou probabilistes,
- politiques définitives ou révisables,
- politiques d'adjudication (avec autonomie de décision aux consommateurs).

3.5 Conclusion

Dans ce chapitre, les différentes composantes logicielles rentrant en jeu dans un environnement de programmation parallèle ont été cernées. Celles qui influencent

le placement, principalement issues du système d'exploitation, ont été décrites. Une série de paramètres basés sur le système et influençant les spécifications de modèles de programme ont été énumérées. Ensuite une description détaillée de deux modèles de programmation statique a été donnée et les autres modèles ont été passés en revue.

Chapitre 4

Résultats existants

4.1 Introduction

Ce chapitre est entièrement consacré à un état de l'art sur le problème de l'ordonnancement et de manière plus approfondie sur le placement de graphes sans relation de précédence. L'accent est mis sur l'intégration d'un procédé de placement automatique dans un environnement de programmation parallèle. La double possibilité d'utilisation des algorithmes de placement est soulignée : si le graphe de départ est sans précédence, ils peuvent être utilisés directement tandis que si le graphe de départ est un DAG, une première phase de regroupement (*clustering*) doit être effectuée.

4.2 Chaîne de compilation

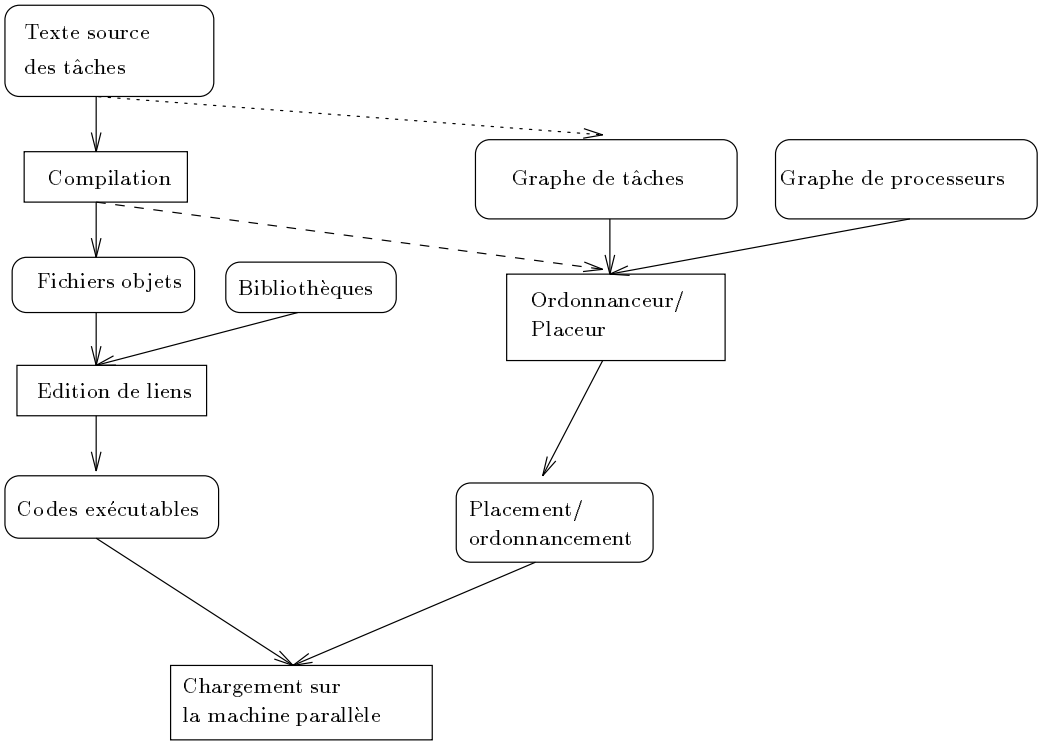


FIG. 4.1 - : Chaîne de compilation

Dans le schéma 4.1, se trouvent décrites les différentes phases existant entre la description du graphe de tâches et l'exécution du programme. La description des tâches peut avoir diverses origines :

- Elle peut être fournie directement par le programmeur. C'est le cas des langages C parallèles existant sur des machines telles que le Méganode.
- Elle peut être dérivée, à l'aide d'outils automatiques, de langages de plus haut niveau (graphiques ou autres).
- Elle peut être extraite de programmes séquentiels à l'aide d'outils de parallélisation automatique (par exemple, FORGE).

La chaîne de transformation d'un programme source en un programme exécutable sur machine parallèle comprend quelques étapes supplémentaires par rapport à la chaîne équivalente sur une machine séquentielle :

- l'extraction, lors de la compilation, des informations permettant de pondérer le graphe de tâches,

- la partie ordonnancement/placement qui prend en entrée le graphe de tâches et le graphe de processeurs et fournit un placement des différentes tâches sur les processeurs (et aussi une date de début d'exécution dans le cas de l'ordonnancement),
- les bibliothèques qui sont liées aux fichiers objets contiennent non seulement les fonctions classiques (mathématiques, gestion de chaînes de caractères, ...), mais aussi une série de fonctions destinées à gérer le parallélisme (sémaphores, primitives de communications synchrones/asynchrones, etc.),
- le chargement du programme qui prend en compte les informations de placement, afin de charger les fichiers exécutables sur les différents processeurs.

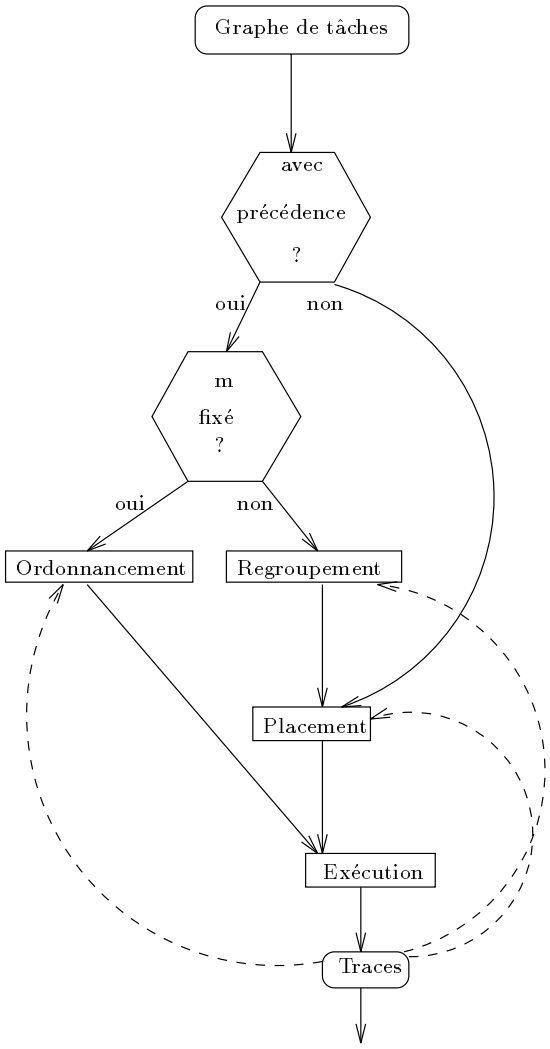


FIG. 4.2 - : Détail du module placement/ordonnancement

Suivant le modèle de programme utilisé trois types de solutions sont utilisées (cf figure 4.2 où m représente le nombre de machines) :

- l'*ordonnancement* qui consiste à partir d'un graphe de tâches avec relation de précédence à fournir pour chacune des tâches un processeur et une date de début d'exécution. Le résultat fourni par un ordonnancement peut être aisément représenté sous la forme d'un diagramme de GANTT ayant le temps sur un des axes et les processeurs sur l'autre (cf figure 4.3),
- le *placement* qui consiste, à partir d'un graphe de tâches sans précédence, à fournir pour chacune des tâches un processeur sur lequel elle va s'exécuter,
- le *regroupement/placement* consiste dans une première phase à regrouper les tâches provenant d'un graphe de précédence et, dans une seconde phase, à placer les groupes sur les différents processeurs. Le regroupement (*clustering*) peut être vu comme le fait de travailler avec une architecture virtuelle de processeurs, c'est-à-dire en considérant que tous les processeurs sont connectés et qu'il y en a un nombre illimité, tandis que le placement consiste à se ramener à une architecture réelle, c'est-à-dire, comportant un nombre fixe de processeurs pas toujours complètement connectés.

4.3 Ordonnancement/regroupement

4.3.1 Définitions et objectifs

Différents types d'objectifs peuvent être envisagés. Le plus courant consiste à vouloir minimiser le temps total d'exécution, C_{max}, aussi appelé *makespan*. D'autres buts tels que la minimisation de l'utilisation des ressources dans le cadre d'une machine multi-utilisateurs sont aussi intéressants. Dans le cadre de l'ordonnancement classique, on retrouve la minimisation des dépassements des dates d'exigibilité, etc. Le but de minimiser le temps d'exécution peut être accompagné de sous-objectifs, tel que la minimisation du nombre de processeurs occupés, ce qui permettra donc de trouver le nombre minimum de processeurs nécessaires qui permet de fournir un ordonnancement qui minimise le temps total d'exécution.

Remarquons que dans le cas où il y a des communications et une clique de processeurs homogènes, le graphe de tâches décrit correspond en fait à un ensemble de graphes de tâches. En effet toutes les communications n'auront pas forcément lieu, celles se passant sur un même processeur auront un coût qui peut être considéré comme nul (un simple passage d'adresse). La notion de chemin critique correspond au cas où chaque tâche est placée sur un processeur différent (toutes les communications sont coûteuses). La notion de séquence dominante [YG92], correspond, elle, au temps total d'exécution de l'algorithme en prenant en compte l'annulation des communications internes (quand les tâches sont placées sur un même processeur).

Un type d'algorithmes très utilisé dans le cadre du problème de l'ordonnancement/regroupement sont les algorithmes de liste.
Un *algorithme de liste* consiste à établir tout d'abord une liste de priorités des tâches suivant certains critères (par exemple, coût de calcul, nombre de fils, etc.) et ensuite à placer les tâches en suivant l'ordre de cette liste sur les différents processeurs. Les listes doivent avoir comme propriété de respecter les relations de précédence inter-tâches ; en fait, un ordre total est constitué à partir de l'ordre partiel des contraintes de précédence. La *compacité*, propriété de certains algorithmes de liste consiste à ne pas laisser un processeur inactif alors qu'il existe une tâche prête à être exécutée.

La plupart des problèmes d'ordonnancement et de placement sont NP-difficiles [GJ79] et la recherche d'une solution optimale (minimisant le temps d'exécution du programme parallèle) demande l'utilisation d'algorithmes de complexité exponentielle. Malheureusement un algorithme de complexité exponentielle est souvent beacoup trop coûteux : par exemple, si l'examen d'une solution prend une micro-seconde, que les entrées sont de taille n=60 et que l'algorithme est de complexité 2^n, son exécution peut prendre jusqu'à 366 siècles.

4.3.2 Résultats d'ordonnancement

On peut distinguer quelques uns des résultats existants suivant les coûts de calcul et de communication :

- Les graphes UET[1] caractérisés par un coût d'exécution unitaire des différentes tâches et par l'absence de coût de communication. Coffman et Graham [CG72] ont trouvé un algorithme polynomial qui permet d'ordonnancer de manière optimale un DAG UET sur deux processeurs. Pour $m \geq 2$ et fixé (où m représente le nombre de machines), le problème reste ouvert (on ne sait pas s'il est NP-complet ou non). A m non borné, le problème devient simple (un solution triviale serait de placer chaque tâche sur un processeur différent). Des résultats existent aussi pour des familles de graphes spécifiques telles que les arbres ainsi Hu a obtenu un résultat d'optimalité en temps polynomial [Hu61] , mais aussi pour les forêts opposées. Appliqué à m machines, l'algorithme de Coffman-Graham possède une borne au pire. Si ω représente la date de fin d'exécution pour l'ordonnancement fourni par cet algorithme et ω_{opt} l'optimum global, la borne au pire est : $\frac{\omega}{\omega_{opt}} \leq 2 - \frac{2}{m}$ [CG72][Bra90].

- Les graphes dont les tâches possèdent un coût de calcul quelconque et pas de coût de communication. L'ordonnancement de tâches indépendantes sur un ordinateur parallèle non-préemptif (pas de pseudo-parallélisme) avec pour objectif, la minimisation du temps total d'exécution est un problème

[1] UET: *Unit Execution Time*

NP-complet[GGJ78]. Plus formellement, soit un ensemble de tâches $T = \{T_1, T_2, \ldots, T_n\}$, chaque tâche T_i possédant un coût de calcul $calc(T_i)$, et un ensemble de processeurs identiques $m \geq 2$. Un ordonnancement, dans ce cas correspond à une partition $Part =< Part_1, Part_2, \ldots, Part_m >$ de T en m ensembles disjoints, un pour chaque processeur. Le i^{eme} processeur, $1 \leq i \leq m$, exécute les tâches de $Part_i$. Le temps total est donné par $f(Part) = \max_{1 \leq i \leq m} calc(Part_i)$ où pour tout $X \subseteq T$, $calc(X)$ est défini par $\sum_{T \in X} calc(T)$. Un ordonnancement optimal $Part^*$ est un ordonnancement qui satisfait à $f(Part^*) \leq f(Part)$ pour toute partition $Part$ de T en m sous-ensembles. Comme il n'y a qu'un nombre fini de partitions possibles, un tel ordonnancement existe mais il n'y a pas d'algorithmes connus de complexité polynomiale qui permette de le trouver. Le problème initial se résume à un problème de *bin packing* qui est connu comme étant NP-complet [GJ79].

- Les graphes UECT [2] où les coûts de communications externes à un processeur sont constants et équivalents aux coûts de calcul des tâches. Une borne a été établie pour les algorithmes de liste fournissant des ordonnancements compacts (un processeur ne reste pas inactif si une tâche est prête à être exécutée) par Rayward-Smith [RS87] : $\omega \leq \left(3 - \frac{2}{m}\right) \omega_{opt} - \left(1 - \frac{1}{m}\right)$. Un algorithme d'ordonnancement optimal pour les arbres a été trouvé par [VRKL94].

- Les graphes possédant des coûts de communication quelconques et des coûts de calcul quelconque. Colin et Chrétienne [CC91] ont montré qu'en permettant la duplication des tâches, avec un nombre non-borné de machines, et avec une hypothèse supplémentaire limitant la taille des communications par rapport aux calculs, on pouvait trouver un algorithme de type chemin critique qui était optimal.

L'hétérogénéité des processeurs peut elle aussi être prise en compte. Mais même dans le cadre de processeurs uniformes (certains processeurs sont uniformément plus rapides que d'autres), peu de résultats existent [BBGT93] [Bae74].

4.3.3 Algorithmes de regroupement

De nombreuses heuristiques de regroupement sont décrites dans la littérature, elles peuvent être subdivisées en deux classes : les regroupements linéaires, où deux tâches non-liées par une relation de précédence (directe ou non) ne peuvent pas appartenir à un même groupe et les heuristiques non-linéaires où il n'y a pas de restrictions quant à la formation des groupes. Un exemple de regroupement linéaire donné par [KB88] consiste à prendre le chemin critique du graphe

[2] UECT: *Unit Execution and Communication Time*

et à regrouper des tâches faisant partie de celui-ci, et procéder ensuite de la même manière pour le nouveau chemin critique obtenu ... Un exemple de regroupement non-linéaire donné par [Sar89] consiste à trier les communications par ordre décroissant d'importance, à annuler la première communication (grouper les tâches la composant) si cela permet de diminuer le temps d'exécution global et procéder de même pour les autres.

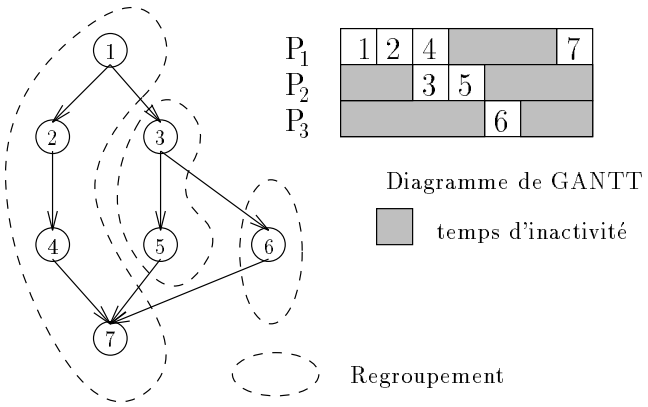


FIG. 4.3 - : Regroupement et diagramme de GANTT

L'algorithme DSC (*Dominant Sequence Clustering*) de Tao Yang et Apostolos Gerasoulis [YG92] est basé sur les trois heuristiques suivantes :

- Le temps total d'exécution est déterminé par la séquence dominante. Dès lors il faut au minimum mettre à zéro (les tâches étant sur le même processeur) un arc appartenant à la séquence dominante.
- Un algorithme de ce type, basé sur la mise à zéro d'arcs de la séquence dominante, peut être fait de manière incrémentale en une série d'itérations.
- la mise à zéro d'un arc est acceptée si le temps total d'exécution diminue grâce à cela.

4.3.4 Avantages/désavantages

Le choix d'un modèle prenant en compte les relations de précédence possède avantages et désavantages :

⊕ Le modèle utilisé représente complètement le comportement du programme modélisé.

⊕ La solution fournie comporte non-seulement une allocation mais aussi une date de démarrage des différentes tâches.

⊖ Un grand nombre de programmes ne sont pas représentables par ce modèle de programme qui ne reprend pas le caractère non-déterministe des programmes parallèles, etc.

⊖ Le problème est NP-complet.

▷ La granularité utilisée dans ce genre de problèmes est généralement faible, ce qui implique que :

- ⊕ Le modèle décrit précisemment le comportement du programme.
- ⊖ Le grand nombre de tâches participe à l'explosion combinatoire des possibilités d'ordonnancement, ce qui a tendance à donner des algorithmes coûteux. Seuls des algorithmes de faible complexité peuvent être utilisés de manière pratique ($< O(n^3)$).
- ⊖ Peu de problèmes liés à l'architecture ou au système d'exploitation se retrouvent lissés. Le modèle de machine devrait donc être aussi précis que le modèle de programme.

4.4 Placement

Nous nous intéressons aux placements statiques, c'est-à-dire réalisés avant l'exécution du programme parallèle. Cette solution n'est envisageable que lorsque l'on connaît à l'avance assez précisément le schéma d'exécution de son programme ; ce qui est le cas dans de nombreux problèmes de calcul numérique par exemple.

Un placement est une application (notée *alloc*) qui, à une tâche, associe un processeur.

$$\forall t \in T, \exists p \in P, alloc(t) = p$$

où T est l'ensemble des tâches à placer et P l'ensemble des processeurs.

La recherche d'un placement se fait sur l'ensemble PL de tous les placements possibles. Si $|P|$ le nombre de processeurs et $|T|$ le nombre de tâches, alors il existe $|P|^{|T|}$ placements possibles, ce qui rend trop coûteux l'exploration de toutes les possibilités de placement (ce problème est connu comme étant NP difficile [GJ79]).
La résolution de ce type de problèmes possède donc avantages et désavantages :

▷ Le modèle représente des programmes ayant une granularité assez forte, ce qui amène deux avantages et un désavantage :

- ⊕ Un nombre de tâches limité permet de limiter l'explosion combinatoire des solutions existantes.

- ⊕ L'expression d'un programme à l'aide d'un grain assez fort permet de décrire le comportement moyen de celui-ci et de lisser certaines particularités du système d'exploitation ou de la machine.
- ⊖ La solution envisagée ne prend pas en compte le comportement fin des programmes.

⊖ Bon nombre de programmes ne sont pas représentables par ce modèle qui ne reprend pas la possibilité de création dynamique de tâches, etc.

⊖ Le problème est NP-complet.

Le but à atteindre avec un bon placement dépend d'une série de critères dont les principaux sont définis ci-dessous. Cette série de critères permet d'établir l'objectif du placement, cet objectif est défini, le plus souvent sous forme d'une fonction de coût à optimiser.

4.4.1 Critères de placement

Dans le cas où le nombre de tâches est inférieur au nombre de machines et où les communications sont moins importantes que le calcul, le problème du placement peut se restreindre au problème du plongement. De très nombreux travaux ont été développés dans le domaine du plongement de graphes et peuvent être employés pour le placement. Un plongement est une application qui associe une arête du graphe de tâches à un chemin dans le graphe cible (ici, le réseau de processeurs) [dR94]. Il existe quelques heuristiques générales, mais la plupart des résultats concernent le passage entre topologies déterminées (par exemple, une grille (représentant une matrice) dans un hypercube de processeurs, ou encore un arbre binaire représentant une expression algébrique dans un anneau ou un tore de processeurs, etc.). Un article de synthèse reprend les résultats connus [MS90]. Usuellement, on considère deux critères : la *dilatation* qui est le maximum des longueurs des plus longs chemins et la *congestion* qui est le nombre maximum de chemins passant par une arête donnée.

Dans le cas $|T| > |P|$ (le plus fréquent en pratique), une solution possible consiste à effectuer une phase préliminaire de regroupement et à travailler alors avec $|P|$ groupes de tâches.

Notons $calc(t_i)$ la duré d'une tâche t_i et $comm(t_i, t_j)$ la durée de la communication de la tâche t_i vers la tâche t_j.

Un placement peut être caractérisé par une fonction de coût qui permet de mesurer sa qualité. Elle apparaît explicitement dans tous les algorithmes de placement basés sur les techniques d'optimisation. Par contre, dans certains algorithmes heuristiques, elle est implicite. Il existe de nombreux choix pour cette fonction de coût, nous nous proposons par la suite de passer en revue celles qui nous paraissent les plus pertinentes.

Un des objectifs les plus désirés est de minimiser le temps total d'exécution du programme qui, sur une machine parallèle, correspond au temps d'exécution du processeur qui termine le dernier. Le calcul du temps d'exécution sur un processeur dépend du modèle d'architecture choisi.

En particulier, certains processeurs ont la possibilité d'effectuer ou non des calculs pendant qu'ils communiquent (on parle dans ce cas de recouvrement des calculs et des communications). D'autre part, des communications peuvent ou non être faites simultanément sur un processeur. Nous considérons ici un modèle sans recouvrement entre calculs et communications et à communications non simultanées. Le coût de l'exécution de la tâche t_i pour le placement *alloc* est calculé de la façon suivante :

$$calc(t_i) + \sum_{tj|alloc(t_i)\neq alloc(t_j)} comm(t_i, t_j)$$

Autrement dit, lorsqu'une tâche t_i est placée sur le processeur $alloc(t_i)$, le coût d'exécution sur celui-ci est augmenté du coût de calcul de la tâche et du coût de ses communications avec les tâches placées sur les autres processeurs (on ne compte évidemment pas les communications entre deux tâches placées sur le même processeur). Le coût cumulé sur un processeur p_k donné est égal à la somme de tous les coûts partiels des tâches allouées sur ce processeur.

$$t_{alloc}(p_k) = \sum_{t_i|alloc(t_i)=p_k} \left[calc(t_i) + \sum_{tj|alloc(t_j)\neq alloc(t_i)} comm(t_i, t_j) \right]$$

Le coût du placement *alloc* complet est le maximum des coûts d'exécution de tous les processeurs, c'est-à-dire :

$$C_{alloc} = max_{p_k \in P}(t_{alloc}(p_k))$$

Le coût du meilleur placement est donc :

$$C^* = min_{alloc \in PL}(C_{alloc})$$

Evidemment, ce critère n'est pas exact. En particulier, comme nous l'avons déjà souligné, l'approximation obtenue en négligeant les précédences peut conduire à des comportements paradoxaux.

Notons que l'on peut facilement modifier cette fonction si l'on désire tenir compte par exemple du recouvrement calcul/communication (à ce moment là, la somme dans le calcul du coût d'exécution d'une tâche devient un calcul de maximum) ou encore des communications simultanées (il suffit alors de pondérer le terme des communications par le degré moyen du graphe). De plus, il est

possible de distinguer les processeurs entre eux dans le cas d'une architecture hétérogène, etc.

Ces fonctions de coût peuvent être affinées :

- Sur un réseau de processeurs formant une clique, la partie communication peut prendre en compte un coût moyen de communication représenté par un temps moyen de communication entre tâches.

- Sur un réseau point-à-point une autre modélisation des communications serait plus adaptée pour tenir en compte les distances entre processeurs. Si le chemin à prendre pour toutes les communications entre deux nœuds donnés est fixe, une modélisation possible serait un modèle linéaire du type $\beta + \delta\tau L$ où β représente un coût de mise en place de la communication (*startup*), δ représente la distance (le nombre de nœuds à traverser), τ représente le temps pris pour communiquer un octet entre deux nœuds voisins et L représente la longueur du message en octets. Ceci pourrait encore être affiné pour tenir compte des perturbations amenées sur les nœuds intermédiaires, la charge du réseau [tron94].

On trouve dans la littérature de nombreuses autres possibilités de fonctions de coût [NT93]. En particulier des critères structurels tels que la cardinalité [Bok81].

4.4.2 Les différentes solutions

Le problème du placement de tâches dans sa généralité étant NP-difficile, il n'existe pas (sauf pour quelques cas simples) d'algorithme plaçant au mieux, en temps polynômial, un graphe de tâches sur un réseau de processeurs [Bok81]. On a donc le choix entre deux grandes alternatives : chercher l'optimal en risquant l'explosion combinatoire ou bien, se contenter d'un placement approché, trouvé en un temps raisonnable. De très nombreuses stratégies de placement de tâches ont été proposées dans la littérature, comme en témoigne la large bibliographie donnée à la fin de cette thèse. Les résolutions possibles peuvent être envisagées de différentes manières. Ainsi, on distingue :

Les *algorithmes exacts* dont le principe repose sur une exploration de toutes les solutions possibles. Cette approche conduit donc à la solution optimale, mais est très coûteuse en pratique et inemployable pour des exemples trop grands.

Les *algorithmes heuristiques* qui conduisent à une solution approchée, eux-mêmes se divisent en deux catégories. D'une part, les algorithmes gloutons qui permettent de construire une solution de proche en proche en partant d'une solution initiale partielle que l'on complète et d'autre part, les *algorithmes itératifs* qui partent d'une solution complète que l'on améliore par transformations élémentaires. Enfin les heuristiques obtenues comme *généralisations d'algorithmes exacts* qui partent d'algorithmes qui ne sont applicables à la base que dans des cas restreints et qui en relaxant certaines contraintes fournissent des solutions sous-optimales mais valables.

Algorithmes exacts

Dans cette classe, on trouve de nombreux algorithmes de complexité exponentielle. Nous pouvons représenter sous forme d'arbre la construction de toutes les solutions possibles (soit tout l'ensemble PL); en partant de la configuration où aucune tâche n'est placée (la racine) vers l'ensemble des tâches placées (les feuilles).

Différentes explorations de cet arbre sont possibles :

- En profondeur d'abord, on construit le plus rapidement possible une solution [Pea90].
- En développant le nœud ayant la plus petite valeur, étant donnée la fonction de coût et les tâches déjà placées (meilleur d'abord) [Sin87] [ST85].
- En développant le nœud ayant le plus grand espoir d'être à la base d'une solution optimale ; en tenant compte des tâches déjà placées et en utilisant une heuristique sous-estimant les tâches restant à placer (A*) [HNR68]. Une bonne heuristique doit pouvoir fournir une sous-estimation aussi proche que possible de la fonction de coût afin de permettre un parcours d'arbre menant le plus directement possible vers l'optimal et doit être aussi simple à calculer que possible afin de pouvoir parcourir rapidement les nœuds intéressés. Dans la majorité des cas, ces objectifs sont contradictoires, c'est pourquoi, le choix d'une heuristique peut se montrer très difficile. Comme exemple d'heuristique, on peut essayer d'effectuer un placement glouton, dont on connaîtrait une borne (telle que la borne de 4/3 du *LPTF (Largest Processing Time First)* [Gra69] [Lee91] dans le cas de tâches indépendantes), sur les tâches qui restent à placer. Un placement LPTF est un placement obtenu en affectant les tâches par ordre de taille décroissant. Quand la prise en compte d'une tâche en vue du placement se fait, on la place sur le processeur dont la date de fin d'exécution (des tâches déjà placée) est la plus proche. Graham a montré que pour m machines homogènes : $\frac{\omega_{lptf}}{\omega_{opt}} \leq \frac{4}{3} - \frac{1}{3m}$ où ω_{opt} est l'optimum global et ω_{lptf} est la solution obtenue par l'algorithme LPTF.

Ces algorithmes fournissent une solution optimale au problème du placement, mais ont une complexité exponentielle dans le pire cas. Il est cependant possible d'arrêter la recherche lorsque l'on a trouvé une solution qui nous paraît satisfaisante. Ces algorithmes, assez coûteux, devraient être utilisés dans le cas d'applications destinées à être installées une fois pour toutes et qui possèdent un petit nombre de tâches.

Il existe d'autres méthodes d'énumération de l'ensemble des solutions et de recherche de l'optimal. On peut, par exemple, découper le problème en sous-problèmes et envisager une recherche par Branch&Bound [PS82].

Certains algorithmes optimaux existent dans des cas restreints :

- Lo [Lo88] a proposé un algorithme basé sur un couplage maximal (donc polynômial) et minimisant le coût de communication entre tâches à condition de respecter les contraintes suivantes : que le nombre de tâches soit inférieur à deux fois le nombre de processeurs et qu'il y ait au plus deux tâches placées par processeur. Cet algorithme fournit un placement optimal si les tâches sont de durées identiques et si les contraintes sont respectées. L'article propose aussi une adaptation de cet algorithme permettant le traitement (non optimal) du cas plus général sans contrainte mais toujours pour des tâches de mêmes durées.

- Les très nombreux travaux développés dans le domaine du plongement de graphes peuvent être employés pour le placement [MS90].

- D'autres algorithmes sont envisageables dans quantité de cas restreints. Par exemple si les tâches sont indépendantes et de coût unitaire, on sait que le *bin-packing* est optimal, etc.

Algorithmes gloutons

Certaines méthodes permettent de trouver un placement grossier rapidement, c'est le cas des algorithmes gloutons. Le principe consiste à construire de proche en proche le placement. Ainsi, l'allocation de la q-ième tâche à un processeur se fait sous un certain critère à partir du placement partiel réalisé sur les (q-1) premières.

Il est clair que l'algorithme glouton est dépendant du choix arbitraire de la première tâche. Leur principal inconvénient est qu'il est toujours possible de trouver une instance qui les mette en défaut.

On trouve dans cette catégorie les algorithmes proposés par [ABPV89] [Lee91] [Paz89], par exemple, LPTF (largest processing time first), pour lequel on connaît une borne théorique dans le pire des cas [Lee91], dont le seul critère est l'équilibrage de charge. Benhamamouch et Plateau [BP89] décrivent un système basé également sur un équilibrage de charge dans le cas où les communications sont négligeables et modifient le critère si elles deviennent importantes. Le principal avantage des algorithmes gloutons est leur coût très avantageux.

A titre d'exemple, l'idée de l'algorithme glouton amical est de placer au fur et à mesure les tâches qui nécessitent le minimum de routage. On part d'une tâche arbitraire et l'on choisit la tâche qui communique le plus avec les tâches déjà placées (en cas d'égalité, on choisit par exemple la tâche dont le coût de communication est minimal). Une fois la tâche déterminée, on choisit le processeur d'accueil en minimisant une fonction de coût partielle. De nombreux autres algorithmes de placement de type glouton reposent sur le critère d'équilibrage de la charge.

Les algorithmes gloutons sont en général peu coûteux, mais en contrepartie, ils ne sont pas très performants. En effet, le choix arbitraire des premières tâches à placer conditionne l'algorithme complet : il n'y a aucune remise en cause des solutions intermédiaires. De plus, vers la fin de l'algorithme, les choix de placement deviennent de plus en plus limités, ce qui fait chuter les performances.

Algorithmes itératifs

Tous les algorithmes itératifs [HK72] et [Paz89] partent d'une solution initiale complète (pas forcément très bonne) que l'on cherche itérativement à améliorer. Cette solution initiale peut, par exemple, être obtenue par un algorithme glouton. Ces algorithmes sont basés sur une fonction de coût. Dans la plupart des solutions existantes, on procède par permutations de tâches en ne retenant que celles qui améliorent la fonction de coût. Notons que des perturbations aléatoires sont nécessaires dans la plupart des algorithmes afin d'éviter les minima locaux.

Recuit simulé

Cette technique est la plus ancienne et de nombreux articles rapportent des résultats et des comparaisons avec d'autres algorithmes [Ber89] [BM88] [HB88] [Paz89].

L'idée de la méthode du recuit simulé provient de l'observation de phénomènes physiques. Elle est basée sur une analogie avec la physique statistique : lorsque l'on souhaite obtenir un métal ayant la structure la plus régulière possible, on utilise la technique dite du recuit. On chauffe le métal et l'on réduit doucement sa température de telle sorte qu'il reste en équilibre tout en refroidissant. Arrivé à une température suffisamment basse, il demeure dans un état d'équilibre correspondant à l'énergie minimale.

A haute température, il y a beaucoup d'agitation thermique ce qui peut localement augmenter l'énergie du système. Ce phénomène se produit avec une certaine probabilité qui diminue avec la température. Cela correspond algorithmiquement à s'autoriser de sortir d'un minimum local de la fonction que l'on cherche à optimiser.

Souvent, l'utilisation dans le cadre du problème du placement de cette approche est présentée de manière intuitive. Il est difficile de la justifier théoriquement par une simple analogie, et surtout de trouver une signification concrète aux paramètres tels que la température. Nous avons choisi une interprétation plus proche du problème à résoudre.

Le placement peut être vu comme un problème d'optimisation d'une certaine fonction de coût. Partant d'un placement initial complet, on va chercher à l'améliorer par un critère local tel que l'échange de paires de tâches par exemple. On accepte un échange si le placement est amélioré dans le cas général et sous une

certaine probabilité qui décroît lors du déroulement de l'algorithme, on s'autorise à choisir un placement plus mauvais. Ceci peut avoir pour effet de sortir d'un minimum local de "l'énergie" (fonction à optimiser).

Les placements générés sont en moyenne assez bons mais d'après la littérature et notre expérience propre, certains inconvénients en limitent l'usage :

- Le temps de calcul élevé, devient vite disqualifiant pour un nombre important de tâches.
- Basé sur une succession d'étapes de modifications aléatoires du placement, son comportement est imprévisible. Chaque exécution donne un placement différent pour un même problème et il peut fournir parfois de mauvais résultats[Ber89].
- Cet algorithme dépend d'un certain nombre de paramètres (température de départ et d'arrivée, vitesse de refroidissement etc..) qui sont difficiles à ajuster et dépendent en fait du placement à effectuer.

Toutes ces remarques nous amènent à considérer le recuit comme coûteux, peu fiable et d'emploi malaisé. Néanmoins, si l'on est prêt à affronter toutes ces difficultés, le recuit simulé donne en général de très bons résultats. L'expérience acquise et les recherches effectuées depuis de nombreuses années permettent d'inférer de bonnes valeurs ou règles pour les différents paramètres. La théorie nous fournit, par exemple, une démonstration de convergence asymptotique si la décroissance de température suit certains critères [MRSV86]. De même l'algorithme de Metropolis se base sur les statistiques pour donner des critères relatifs à la température initiale et à l'acceptation de la solution.

Déclarations et algorithme du recuit simulé

Nom	Type	But
heat	réel	Température du recuit
minimum	réel	Température finale
local_equilibrium	booléen	L'équilibre local est-il atteint??
move	{(task,dest)}	Tranformations du placement actuel
generate_move	fonction	Choix aléatoire d'un voisin
cost_move	réel	Coût de la solution générée
get_cost	fonction	Estimation de l'amélioration
gen_unif(0,1)	fonction	Génération d'un nombre aléatoire entre 0 et 1
make_move	fonction	Déplacement vers le placement choisi
update_le	fonction	Mise à jour du test d'équilibre local

```
initialize(heat);
generate_starting_solution(cur_sol);
```

```
while (heat > minimum) {
      local_equilibrium=FALSE;
      while (notlocal_equilibrium){
            move=generate_move(cur_col);
            cost_move=get_cost(move);
            if (cost_move < 0)
               make_move(move);
            else
               if ((gen_unif(0,1) <= (1/exp(-cost_move/heat)))
                  make_move(move);
            update_le(local_equilibrium);
            };
      heat=heat*decrease;
      };
```

L'algorithme de Bokhari

L'algorithme de Bokhari [Bok81] utilise une fonction de coût (la cardinalité) qui compte le nombre d'arêtes du graphe de tâches correctement placées sur le graphe des processeurs. On peut l'adapter avec une fonction de coût plus réaliste prenant en compte tous les paramètres (calcul, communication). Plus précisément, l'amélioration du placement s'effectue par échanges itératifs de paires de tâches. Le test d'arrêt est basé sur le principe qu'aucune amélioration n'a été obtenue pendant un nombre arbitraire de tentatives successives. Notons que cet algorithme est d'emploi peu aisé car il repose sur l'hypothèse restrictive que le nombre de tâches est égal au nombre de processeurs. Il ne s'utilise donc que dans le cas de réseaux points-à-points où la notion de distance inter-processeurs n'est pas binaire. Pazat [Paz89] étudie une extension à un nombre de tâches supérieur par repliage du graphe de tâches. A notre sens, cette hypothèse n'est pas limitative car il suffit de considérer directement que l'on peut placer plusieurs tâches par processeurs. PYRROS utilise aussi l'algorithme de Bokhari : dans une première phase le nombre de groupes de tâches est réduit au nombre de processeurs grâce à un algorithme de liste qui trie les groupes tâches par ordre de coût croissant et ensuite les fusionne ; puis l'algorithme de Bokhari est appliqué.

Les algorithmes génétiques

Les algorithmes génétiques [MGSK87] [TB91] sont basés sur une analogie avec l'évolution des espèces. Ils partent d'une population initiale à partir de laquelle ils effectuent des croisements pour engendrer de nouvelles configurations. Ils gardent les meilleures populations résultantes pour effectuer d'autres croisements. Afin d'éviter les minima locaux, de temps à autre, les espèces sont soumises à des

mutations.

Comme pour le recuit, bien que l'analogie puisse paraître assez superficielle, cette technique conduit parfois à de bons résultats comme par exemple dans le cas où le graphe de tâches n'est pas très différent de la topologie physique (par exemple dans le cas d'éléments finis dans une grille). Ces algorithmes sont de même assez difficiles à mettre en œuvre.

La recherche tabu

Au cœur de la recherche *tabu* [GL92], on retrouve une méthode itérative qui recherche, à partir d'une solution initiale, la meilleure solution que l'on peut atteindre à l'aide d'un ensemble donné de transformations (relations de voisinage). Autour de cette recherche, une liste impressionnante de conseils et d'heuristiques a été développée par Fred Glover et Manuel Laguna. Par la suite, de nombreux développements destinés à résoudre des problèmes particuliers ont aussi été mis en œuvre. Voici les principales composantes d'une recherche tabu :

- Une fonction de coût doit permettre d'estimer et de comparer la qualité des différentes solutions explorées.
- Les relations de voisinage permettent de se promener dans l'espace des solutions, de solution en solution. Cette promenade devrait permettre le passage par un optimum global, ou au moins de passer par des minima locaux intéressants.
- A chaque itération, l'ensemble des candidats atteignables à l'aide des relations de voisinage est examiné. Plus les relations de voisinage sont étendues, plus on aura de chances de trouver une bonne solution, mais au prix d'une augmentation du coût d'évaluation de chaque itération.
- Une mémoire est utilisée afin de retenir la meilleure solution trouvée.
- Afin d'éviter de boucler (de revenir sur une solution déjà explorée), une mémoire des solutions parcourues est mise en place. Il s'agit du concept de *liste tabu*. La liste tabu possède une taille limitée et chaque élément de la liste représente une série d'attributs caractérisant la solution représentée ou plutôt caractérisant le déplacement à faire de la situation actuelle vers la situation précédente. Ces attributs doivent être choisis avec soin. Différents cas peuvent être envisagés :
    - enregistrer une description complète de la solution, ce qui peut être coûteux en place mémoire,

  - enregistrer certains attributs peut demander un recalcul coûteux afin de comparer les solutions (par exemple le calcul de la fonction objective),
  - au cas où le problème comporte de nombreuses solutions qui ne peuvent être distinguées (entre autres, par la fonction objective), les attributs à enregistrer devraient permettre de cataloguer une famille de solutions semblables et non plus une solution précise.

- Le *critère d'aspiration* doit permettre de revenir sur une solution qui n'est pas admise par la liste tabu. En effet la liste tabu ne représente qu'une vue partielle et par exemple, le dépassement de la valeur de la fonction objective de la meilleure solution jamais rencontrée peut être considéré comme critère d'aspiration.

- Les *critères d'arrêt* peuvent être multiples. Par exemple, on peut s'arrêter si une solution satisfaisante est rencontrée, ou au bout d'un certain temps, ou encore, si aucune amélioration de la meilleure solution rencontrée n'a été trouvée depuis longtemps.

- Les fonctions *d'intensification* et de *diversification* permettent de s'attarder ou de s'éloigner de voisinages qui semblent ou ne semblent pas intéressants.

Il existe trois cas où une description considérée à un moment comme tabu peut être à nouveau explorée :

- La solution considérée est sortie de la liste tabu qui possède une taille limitée. Ce cas peut indiquer le fait que la liste tabu est trop courte.

- La solution appartient à la liste tabu et le critère d'aspiration est satisfait.

- Tous les éléments visitables aux différents voisinages sont contenus dans la liste tabu. Ceci peut indiquer le fait que les fonctions de voisinage sont trop restreintes, ou que les attributs enregistrés dans la liste tabu sont trop vagues et représentent par là trop de solutions.

Les recherches *tabu* sont en général plus difficile à mettre en œuvre que les recuits simulés, mais constituent actuellement les algorithmes itératifs donnant les meilleurs résultats, entre autres dans les problèmes de type job-shop[Len93]. Dans la recherche tabu, l'expérience acquise et la connaissance du problème permettent de fixer de bonnes valeurs aux différents paramètres. Les relations de voisinage vont découler de l'étude du problème, tandis que la taille de la liste tabu sera soit fixée arbitrairement soit issue de l'expérimentation (on fixe tout d'abord une petite taille que l'on augmente jusqu'au moment où le tabu permet d'éviter tous les cycles).
Les enregistrements effectués dans la liste tabu ne sont pas toujours complets :

on peut tomber sur une situation que la liste tabu considère comme connue alors qu'elle ne l'est pas... Dans ce cas, la mise en œuvre de ce qu'on appelle un critère d'aspiration s'impose. Un exemple simple consiste à examiner si la nouvelle situation améliore la fonction objective par rapport à la situation précédemment rencontrée. Si oui, le voisin est accepté, si non, on considère qu'il s'agit bien d'une situation déjà rencontrée et on n'accepte pas le voisin.
D'autres fonctions d'intensification ou de diversification de la recherche peuvent aussi être utilisées suivant le comportement de l'algorithme.

Déclarations et algorithme de recherche tabu

Nom	Type	But
tabu_continue	booléen	La recherche tabu continue?
find_best_neighbor	fonction	Donne le meilleur voisin du placement
best_move	{(task,dest)}	Le meilleur voisin
neighboring	fonction	Génère de manière successive le voisinage
it_is_not_tabu	fonction	Vérification de la liste tabu
aspiration_criterion	fonction	Vérification du critère d'aspiration
update_tabu_list	fonction	Ajout dans la liste tabu
update_cont	fonction	Mise à jour de l'intérêt de continuer la recherche

Recherche tabu

```
optimize=TRUE;
generate_starting_solution(cur_sol);
tabu_continue=TRUE
while (tabu_continue) {
      best_move=NULL
      for each move in neighboring(cur_sol) {
               if ((get_cost(move)<get_cost(best_move))
                   and (it_is_not_tabu(move))
                   or aspiration_criterion(move))
               best_move=move;
               };
      make_move(best_move);
      update_tabu_list(best_move);
      update_cont(tabu_continue);
      };
```

Algorithmes exacts généralisés

De nombreux algorithmes exacts existent dans des cas très restreints et sont utilisés de manière sous-optimale en les généralisant en relaxant certaines contraintes. Dans cet ordre d'idées, citons entre autres :

- Pellégrini et Roman [PR93] ont développé une heuristique de type *divide and conquer* qui effectue un placement par bi-partition récursive du graphe de tâches, sous-tendue par une bipartition récursive du graphe de processeurs. Ceci est à rapprocher des solutions par coupes minimales de graphes bi-partis[Lo84] [Sto77].

- Certains dérivés d'algorithmes d'exploration d'arbres de recherche peuvent couper certaines branches de l'arbre de recherche de manière heuristique et arbitraire afin de trouver une solution approchée en un temps acceptable.

- Certains résultats issus de la théorie du plongement peuvent être utilisés sur des topologies qui s'approchent de celles conseillées.

4.4.3 Autres solutions et conclusion

L'intégration des outils relatifs au placement de tâches dans un environnement de programmation parallèle a été décrit dans ce chapitre. Un état de l'art des solutions données aux problèmes d'ordonnancement et plus particulièrement des algorithmes destinés et utilisables pour le problème du placement a été dressé. Une double utilisation des algorithmes de placement a été donnée : non seulement en tant que solution à part entière pour des graphes de tâches sans précédence, mais aussi comme moyen de projeter sur une architecture réelle un graphe de tâches avec relation de précédence qui aurait été regroupé (et utilise donc autant de processeurs virtuels que désirés).

Le grand nombre de méthodes que nous venons de présenter montre qu'il n'y a pas de solution universelle au problème du placement de tâches. L'approche la plus prudente est de développer une boite à outils de méthodes de placement, utilisable par le programmeur au cas par cas. En particulier, il est possible de considérer des stratégies intermédiaires entre les différentes solutions que nous avons proposées, par exemple, à mi-chemin entre les heuristiques gloutonnes et un Branch&Bound.

On peut voir cette approche mixte comme un algorithme heuristique dont certains choix peuvent être remis en cause et un certain nombre de placements sont alors systématiquement passés en revue. On peut aussi l'envisager comme un Branch&Bound dont le nombre de combinaisons est très fortement limité par une stratégie de choix des branches visitées.

Les stratégies mixtes sont très prometteuses. En effet, idéalement, la remise en cause de certains choix permet d'éviter l'existence de cas pathogènes qui sont

propres aux algorithmes où des décisions arbitraires ont été prises. Inversement, la limitation de l'explosion combinatoire donne à cet algorithme un temps de réponse acceptable par rapport à n'importe quel Branch& Bound simple. Kasahara et Nahira [KN84] présentent des expérimentations avec un nombre de tâches de l'ordre de 200.

Malheureusement, il n'existe à notre connaissance aucune étude approfondie de ce type d'algorithme.

Une autre solution envisagée serait d'effectuer un premier placement avec un algorithme glouton et ensuite de l'améliorer localement, là où se situent certains problèmes ; par exemple, on pourrait découvrir, suite à une analyse effectuée après une exécution par des outils de prise de traces, qu'un lien de communication entre deux processeurs est surchargé et essayer de remédier à cela en tenant compte d'une fonction de coût (équilibrage de charge totale). De tels outils constituent déjà une ouverture vers des approches dynamiques.

Chapitre 5

Une plate-forme d'aide au placement

5.1 Introduction

Les chapitres précédents vont nous aider à choisir une série de modèles qui seront à la base d'une solution originale au problème du placement. Cette solution s'appuie sur (et est interfacée à) un environnement de programmation complet. Trois types d'algorithmes (gloutons, itératifs et exacts) ont été conçus et implémentés. Parmi ceux-ci, on retrouve plus particulièrement un recuit simulé et une recherche tabu. Ces algorithmes optimisent différentes fonctions objectives (des plus simples et universelles aux plus complexes et ciblées). Dans le cas où le graphe de tâches est de grande taille et où le placement effectué n'est pas primordial, des algorithmes simples sont à envisager (gloutons). Tandis que si le graphe de tâches comporte un petit nombre de nœuds (i.e. ≤ 16), des algorithmes exacts peuvent être utilisés. Dans le cas général, la solution proposée consiste à effectuer un premier placement à l'aide, par exemple d'un algorithme glouton, et à affiner celui-ci après une première exécution, en effet à ce moment là, des outils de prises de mesures auront permis de mieux connaître les différents paramètres.

5.2 Objectif

Trois demandes furent principalement à l'origine du présent travail :

- La première demande qui fut à l'origine de cette thèse émana d'un constructeur français, Archipel SA dans le cadre d'un contrat AASI avec le MRE. Cette société désirait intégrer un outil de placement automatique à son environnement de programmation, VOLCONF, destiné aux réseaux de transputers. Un des objectifs de cet outil était d'améliorer les performances par rapport à un placement aléatoire mais aussi de perturber le moins possible

les habitudes de programmation de leurs clients. Le langage destiné à être enrichi pour supporter le placement était le langage C d'Inmos.

- Une deuxième demande fut celle émise au sein du projet européen Copernicus, sous-projet SEPP (*Software Engineering for Parallel Processing*). Cette proposition souligna le manque de généralité de la représentation sous forme de graphes de précédence et appela à une représentation sous forme de graphes sans relation de précédence.

- La troisième demande est interne à l'équipe de calcul parallèle (projet APACHE au sein du LMC-IMAG et du LGI-IMAG) qui a développé et développe un environnement de programmation parallèle avec plusieurs objectifs. Parmi ceux-ci on retrouve l'étude du problème du placement-ordonnancement, des communications dans les réseaux de processeurs, de l'évaluation des performances des machines parallèles, du portage d'applications numériques et de calcul formel sur machines parallèles, de langages de haut niveau (prolog) sur ces machines, etc.

La demande interne peut être vue en deux phases correspondant au développement de deux environnements de programmation distincts destinés aux machines parallèles.

Un premier environnement appelé OUF (*Occam User's Frustration*) était destiné à remédier à la profusion des langages C parallèles et des routeurs destinés aux machines à mémoire distribuée. Les graphes de tâches décrits par les langages C parallèles disponibles sur réseaux de transputers (Logical C, C3L, Inmos C) ressemblent beaucoup aux graphes de tâches sans précédence décrits dans la sous-section 3.4.3. L'objectif d'un tel langage, qui peut être vu principalement comme une interface abstraite, était de garantir la portabilité des programmes existants et de pouvoir profiter des performances et possibilités des nouveaux langages. Dans le graphique 5.1, deux instances classiques de OUF sont représentées : OUF destiné aux réseaux de transputers utilisant Inmos C et VCR, et OUF destiné aux réseaux locaux utilisant PVM et une bibliothèque de processus légers.

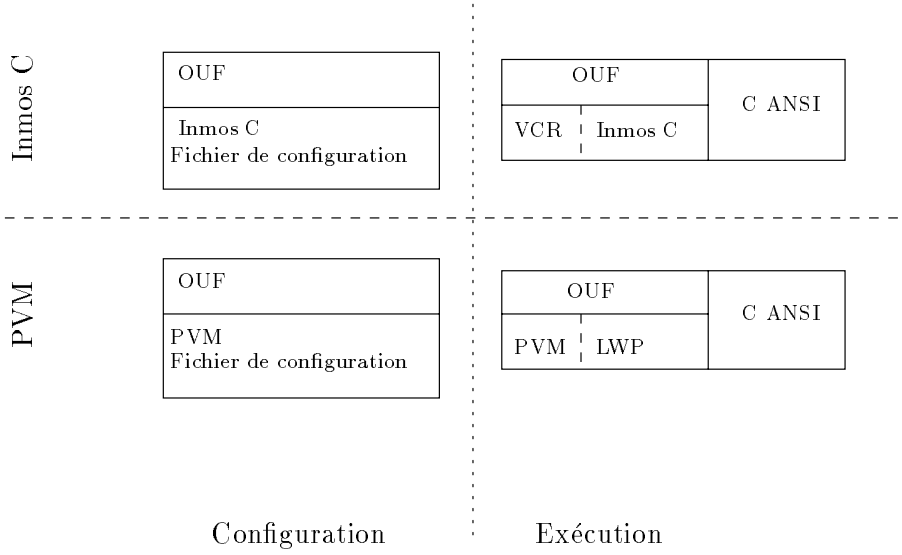


FIG. 5.1 - : Deux instances de OUF

Un deuxième environnement, basé sur le langage Athapascan est développé par le groupe Apache(cf figure 5.2, [BCR94]). Ce langage est basé sur la notion de RPC [1] asynchrone. Les ajouts à apporter à un système d'exploitation standard (Unix) afin d'exploiter de manière performante une machine parallèle sont généralement une bibliothèque de gestion de processus légers (LWP de Sun, Bkernel de J. Briat, etc.) et une bibliothèque de gestion du parallélisme dont les communications et le placement (VCR ou Tiny comme routeurs, PVM ou MPI comme gestion plus étendue du parallélisme).

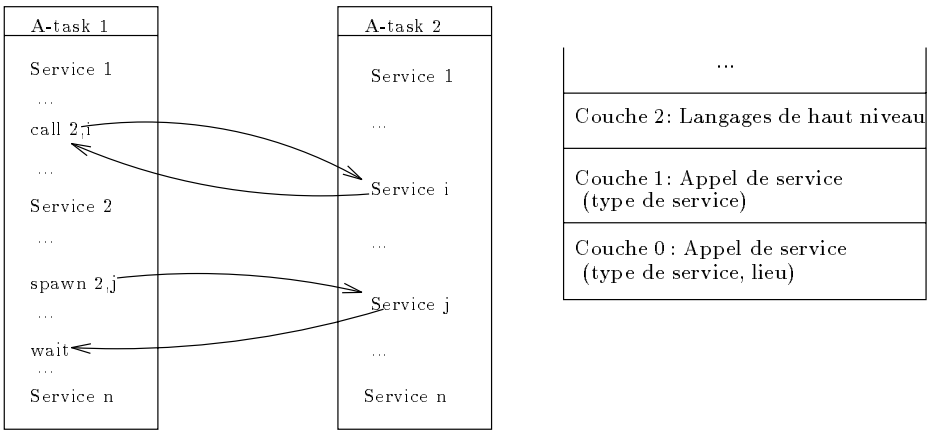


FIG. 5.2 - : Appel de service et découpe en couches de Athapascan

[1] RPC: *Remote Procedure Call*

5.3 Modèle de machine

L'évolution des machines actuelles montre que le concept qui tend à se généraliser est celui des machines parallèles à mémoire distribuée interconnectées à l'aide de réseaux multi-étages (SP2 d'IBM, CS2 de PCI, CM5 de TMC) ou de réseaux fixes améliorés à l'aide de systèmes de routage rapides (T3D de Cray, Paragon d'Intel). Cette nouvelle génération amène à considérer que tous les processeurs sont à même distance. En effet soit la communication est interne et très peu coûteuse, soit elle est externe et les coûts de latence logicielle sont très élevés et masquent les coûts de latence matérielle.

Par exemple sur la SP1 d'IBM, pour une machine ayant un nombre fixé de processeurs, une communication entre deux processeurs quelconques de la machine passera toujours par le même nombre de *switches*. Sur la CM5 de TMC, la structure *fat-tree* de son réseau d'interconnexion fait que la hauteur de remontée dans l'arbre des communications dépend de la proximité des processeurs, mais la latence logicielle masque quasi complètement ce phénomène.

L'accent a été mis, dans la solution proposée, sur l'aspect répartition de la charge et la dichotomie communication interne versus communication externe. La notion de distance inter-processeurs n'a été ajoutée que pour tenir en compte l'architecture des machines possédant des architectures proches des réseaux de transputers.

Deux types de machines parallèles sont directement disponibles au laboratoire LMC-IMAG et sont donc directement concernées par les outils de placement:

- Le Méganode, machine à 128 transputers connectés via leurs liens et divers *switches*. Chaque nœud possède 1 méga-octet de mémoire vive et 4 connexions vers d'autres nœuds.

- Un IBM SP1, constitué de 32 nœuds avec 64 méga-octets de mémoire et 1 giga-octet de disque. Chaque nœud est connecté à un réseau Ethernet destiné à considérer les nœuds comme un réseau local classique et est connecté aussi à tous les autres nœuds via un réseau multi-étages.

5.4 Modèle de programme

Les graphes de tâches sans relation de précédence ont été choisis comme modèles pour différentes raisons pré-citées que nous rappelons ici :

- ils permettent de modéliser plus de problèmes que les graphes acycliques de précédence,

- ils sont très proches des fichiers de description des langages C parallèles,

- ils peuvent aider, suite à un regroupement effectué sur un graphe de précédence, à se ramener d'une architecture virtuelle à une architecture réelle.
- Ils se situent à un niveau de granularité suffisamment grand pour atténuer les problèmes de modélisation matériels et logiciels.

5.5 Fonction objective

Diverses fonctions de coût ont été prises en compte. La fonction de coût de base tente d'établir un compromis entre minimisation des communications et équilibrage de la charge de calcul (appelons cette fonction SUM).

$$max_{p_k \in P} \sum_{t_i | alloc(t_i)=p_k} \left[calc(t_i) + \sum_{tj | alloc(t_j) \neq alloc(t_i)} comm(t_i, t_j) \right]$$

Les données fournies sont calculées en moyenne et en unités de temps. Un moyen d'estimer en moyenne la durée des communications est de multiplier le nombre d'octets communiqués entre deux processeurs quelconques par le rayon (la moitié du chemin le plus long existant entre deux processeurs) du graphe de processeurs et par le temps mis par la communication d'un octet entre deux processeurs voisins.

Les algorithmes de placement peuvent aussi prendre en compte une fonction de coût qui considère que communications et calculs peuvent avoir lieu en même temps (grâce à la présence de processeurs spécialisés dans les communications, appelons cette fonction MAX) :

$$max_{p_k \in P} \max_{t_i | alloc(t_i)=p_k} \left[calc(t_i), \sum_{tj | alloc(t_j) \neq alloc(t_i)} comm(t_i, t_j) \right]$$

Afin de prendre en compte les distances inter-processeurs sur le Méganode, une fonction de coût utilisant les informations contenues dans les tables de routage VCR a été définie. En effet VCR est complètement déterministe et ses tables sont disponibles à la compilation sous forme de fichier ASCII. Voici la fonction de coût prise en compte dans ce cas (appelons cette fonction $ROUT$) :

$$max_{p_k \in P} \sum_{t_i | alloc(t_i)=p_k} \left[calc(t_i) + \sum_{tj | alloc(t_j) \neq alloc(t_i)} \delta_{alloc(t_i), alloc(t_j)} \tau L(t_i, t_j) \right]$$

où $\delta_{a,b}$ représente le nombre de sauts (la distance en termes de processeurs) à effectuer entre le processeur a et le processeur b.

Une fonction plus précise au niveau du calcul du coût des communications a aussi été utilisée. Cette fonction considère que le réseau de processeurs sur le méganode forme un tore 4x4 et utilise des coûts de mise en place des communications et des coûts de passage à l'octet. Le modèle de communication est donc ici linéaire de type $\beta + \delta \tau L$ où les β, coûts de mise en place, et les τ, coûts de passage ont été calculés de manière expérimentale sur le Méganode (appelons cette fonction TOR).

Une dernière fonction de coût est en cours de validation. Cette fonction prendra en compte non seulement les caractéristiques propres à un réseau et à un routeur donné (tore et VCR), mais aussi propres à la charge supportée par ce réseau (bruit).

5.6 Spécification des algorithmes implémentés

5.6.1 les gloutons

L'algorithme modulo

L'algorithme modulo est un des plus simples imaginables. Il consiste à placer la i^{eme} tâche sur le i *modulo* m processeur. Cet algorithme a cependant été amélioré pour tenir compte des contraintes mémoires.

Avantages/Désavantages :

⊕ Cet algorithme est très rapide ($O(n)$).

⊕ Si les tâches sont toutes environ de même coût, cet algorithme établit une bonne répartition de la charge de calcul.

⊕ Si la numérotation logique des tâches fait que les tâches de même poids se suivent au niveau numéro, cet algorithme établit un bon équilibrage de la charge.

⊖ Cet algorithme ne prend en compte ni les coûts de calcul, ni les coûts de communication. Ce qui le rend mauvais dans les cas où :

- Il y a un grand écart type entre les différents calculs.
- Les communications sont importantes car aucune tentative d'effectuer ces communications de manière interne ne sera faite.

⊖ Si $|T| > |P|$, cet algorithme utilise, dans tous les cas, l'ensemble des processeurs fournis. Ce qui peut être un point négatif dans les systèmes multi-utilisateurs où les processeurs inutilisés par une application peuvent être attribués à une autre application.

Algorithme de liste : Plus grand coût de calcul d'abord (LPTF)

LPTF est un algorithme de liste qui essaye de répartir le coût de calcul entre les différents processeurs.

Les tâches sont tout d'abord triées par ordre décroissant de coût de calcul. Ensuite la première tâche est placée sur le premier processeur et la i^{eme} tâche sur le processeur le moins chargé au niveau calcul.

Avantages/Désavantages :

⊕ Cet algorithme est très rapide ($O(n\ log(n))$).

⊕ Cet algorithme établit une bonne répartition de la charge de calcul, qui est au pire à $\frac{4}{3}$ de l'optimal.

⊖ Cet algorithme ne prend pas en compte les coûts de communication. Ce qui le rend donc mauvais dans les cas où les communications prédominent car aucune tentative d'effectuer ces communications de manière interne ne sera faite.

⊖ Si $|T| > |P|$, cet algorithme utilise, dans tous les cas, l'ensemble des processeurs fournis.

Algorithme de liste : Le plus grand coût total d'abord (LGCF)

LGCF (*Largest Global Cost First*) est un algorithme de liste qui essaye de répartir le coût total (calcul & communication) entre les différents processeurs.

Les tâches sont tout d'abord triées par ordre décroissant de coût global (en prenant en compte les coûts de calcul et communication, en considérant que toutes les communications sont externes et donc coûteuses). Ensuite la première tâche est placée sur le premier processeur et la i^{eme} tâche sur le processeur qui minimisera le coût global, représenté par le processeur le plus chargé :

$$max_{p \in P} \sum_{t|alloc(t)=p} \left(calc(t) + \sum_{t'|alloc(t') \neq p} comm(t, t') \right)$$

Avantages/Désavantages :

⊕ Cet algorithme est très rapide ($O(n\ log(n))$).

⊕ Cet algorithme établit une bonne répartition de la charge de calcul et une minimisation des communications.

Dans la figure 5.3, on voit l'évolution de la fonction de coût global (avec coûts de communications) lors du placement des différentes tâches de l'algorithme de Strassen (multiplication de matrices rapide) avec l'algorithme LPTF et l'algorithme LGCF. On peut remarquer que l'algorithme LGCF prend en compte le coût global des tâches et place les plus coûteuses en premier et en dernier celles qui influencent peu le coût total. Tandis que LPTF ne prend pas en compte les coûts de communication, on voit qu'il reste vers la fin des tâches possédant un

grand nombre de communications qui vont dégrader très nettement la solution obtenue.

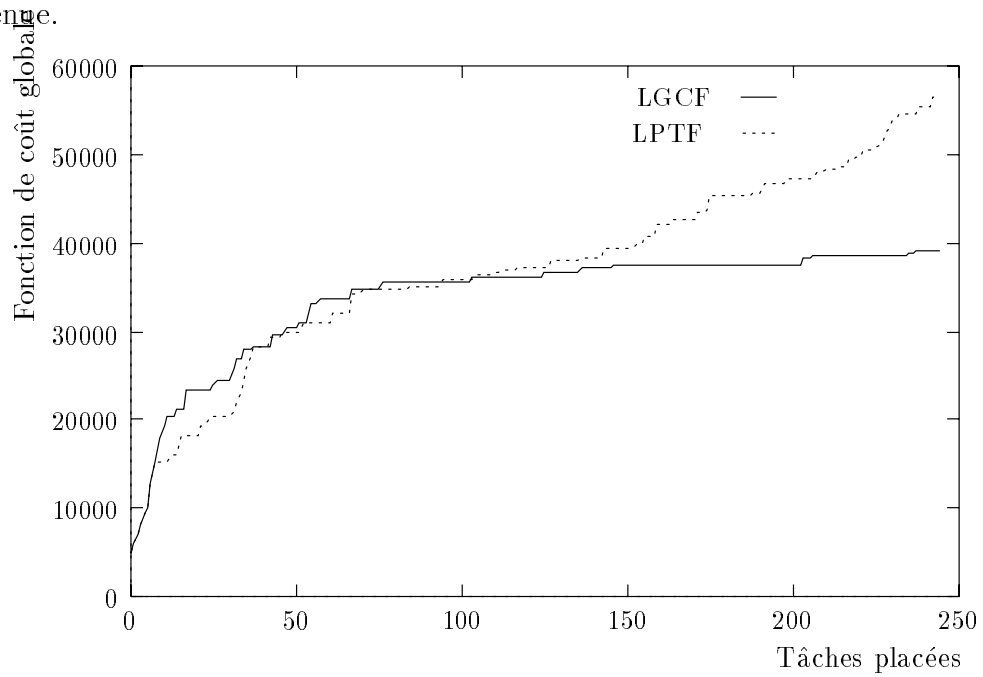


FIG. 5.3 - : Evolution du placement des différentes tâches de l'algorithme de Strassen par LPTF et LGCF.

Algorithme de liste : Critère Structurel (Struc)

Dans les algorithmes précédemment décrits, seuls des critères quantitatifs étaient pris en compte. Des critères structurels peuvent aussi être considérés tels que le nombre de liens de communications sortant de chaque tâche. Les tâches sont triées par ordre décroissant de liens de communications et ensuite sont placées sur le processeur le moins chargé globalement (calcul et communications).

Algorithme de liste : Critère mixte (Struc_quanti)

Des critères mixtes peuvent aussi être considérés. Par exemple, dans l'algorithme proposé, les tâches sont tout d'abord triées par ordre décroissant de liens de communications. Ensuite parmi les tâches possédant un même nombre de liens, un tri est effectué par ordre décroissant de coût global. Finalement les tâches sont placées de manière gloutonne sur le processeur le moins chargé globalement (calcul et communications).

5.6.2 Les algorithmes itératifs

Dans les algorithmes suivants deux types de fonctions de voisinage sont utilisées. Le premier consiste à transférer une tâche du processeur le plus chargé vers

un autre processeur. Le deuxième consiste à échanger une tâche résidant sur le processeur le plus chargé avec une tâche communiquant avec celle-ci et résidant sur un autre processeur. Remarquons que le calcul du coût de transfert d'une tâche va dépendre de la fonction de coût. En effet dans le cas de fonctions de coût simples telles que SUM et MAX, seules les charges des processeurs source et destination doivent être recalculées ; tandis que le cas de fonction plus complexes telles que $ROUT$ et TOR, les charges de tous les processeurs possédant des tâches communiquant avec la tâche à bouger sont à recalculer. En effet les distances par rapport à la tâche vont être modifiées et l'étendue (le caractère plus ou moins local) des transformations effectuées va donc influer sur la complexité de l'algorithme itératif de placement.

Le recuit simulé

Les paramètres suivants ont été déterminé à l'aide de la littérature existante et d'un grand nombre d'expérimentations.

Il existe des preuves de convergence du recuit simulé vers une solution optimale, dans le cas où la température décroît très lentement [MRSV86]. Le graphique 5.4 compare une décroissance du type de celles utilisées dans les démonstrations de convergence ($\frac{1}{log(n)}$) et une décroissance du type de celles utilisées dans la pratique ($n^{0.98}$ où n représente le nombre d'itérations). On peut ainsi remarquer que ce qui est utilisé est bien loin de la théorie.

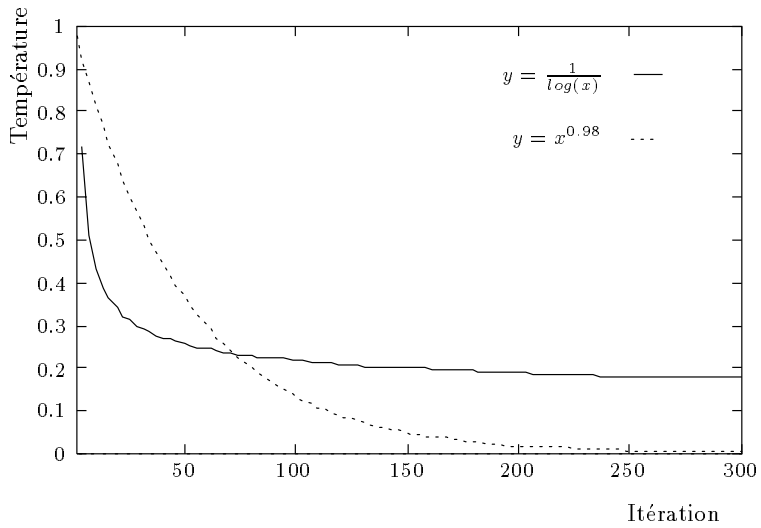


FIG. 5.4 - : Comparaison entre descente de température conseillée par la théorie et par la pratique

Le recuit simulé (cf la section 4.4.2) estime le coût moyen des mouvements (δ_f). Ce δ_f et τ, un pourcentage de départ d'acceptation des mauvaises solutions

(i.e. $\frac{80}{100}$), sont utilisés pour trouver une température de départ ($h = \frac{\delta_f}{log\left(\frac{1}{\tau}\right)}$).

Une mauvaise solution est acceptée si et seulement si $e^{\frac{-\delta_{f_{ij}}}{h}} \geq gen_unif(0,1)$ (cf figure 5.5). Rappelons que $gen_unif(0,1)$ génère un nombre aléatoire de manière uniforme entre 0 et 1.

La figure 5.6 illustre la relation qui existe entre qualité de solution et température initiale (fixée à l'aide d'un taux d'acceptation et du coût moyen de transition) et aussi entre le temps pris pour trouver la solution proposée et la température de départ.

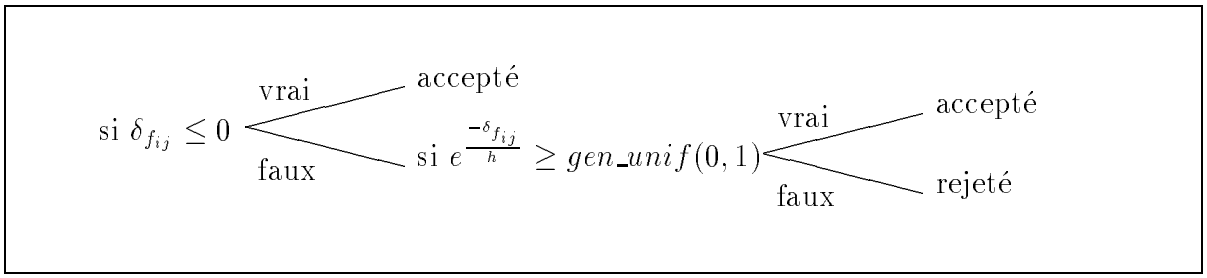


FIG. 5.5 - : Acceptation ou rejet d'un mouvement de coût $\delta_{f_{ij}}$

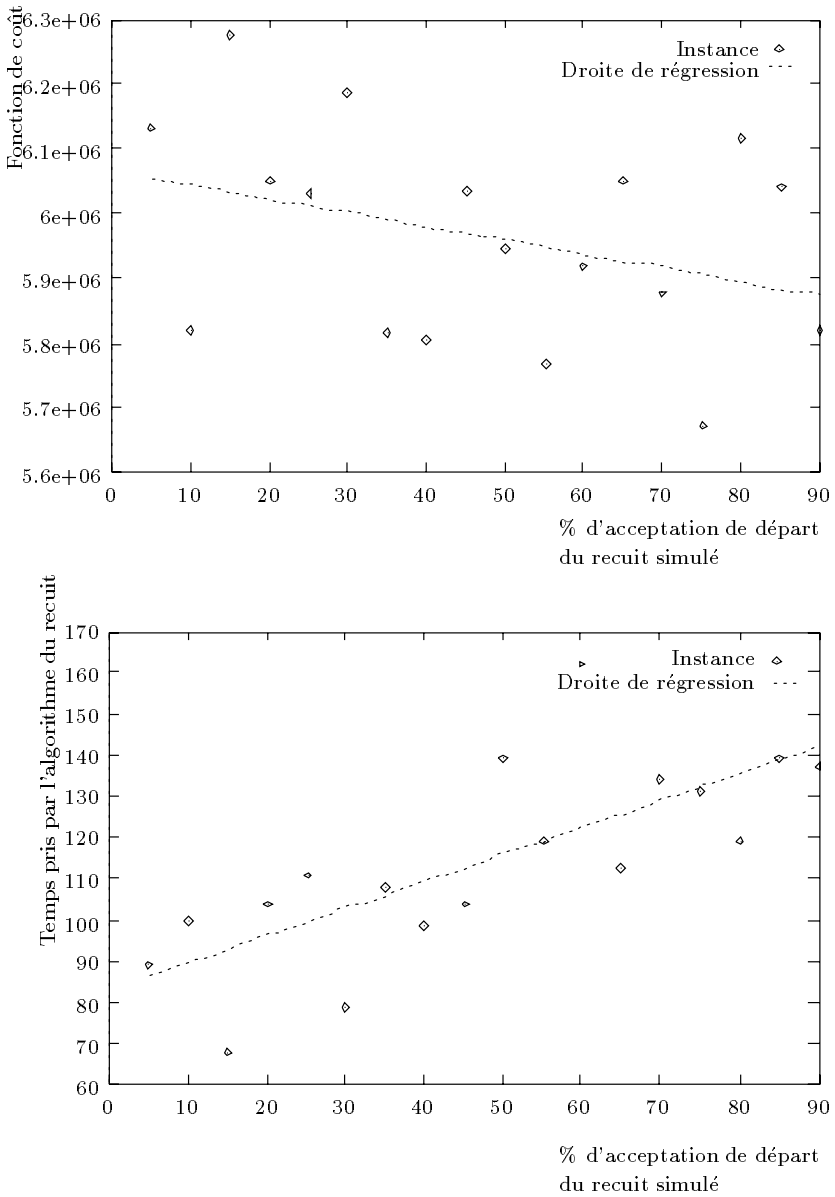


FIG. 5.6 - : Comparaison entre température initiale et qualité de solution

Dans la figure 5.7, nous voyons nettement que l'agitation de départ est très grande et que le recuit oscille entre de très bonnes solution et de très mauvaises, puis par la suite cela se calme et le recuit a tendance à converger vers la meilleure solution. Afin de préserver la meilleure solution obtenue, une mémoire est utilisée et la solution fournie n'est pas la solution finale, mais bien la solution contenue dans cette mémoire.
Deux critères d'arrêt sont utilisés :

- Au bout d'un certain nombre d'itérations n'ayant pas accepté de modifications (100*nombre de tâches), le recuit simulé est interrompu et la meilleure

solution trouvée est fournie.

- L'utilisateur peut fournir à l'algorithme un temps maximum disponible. Au bout de cet temps, le recuit simulé est interrompu et la meilleure solution trouvée est fournie.

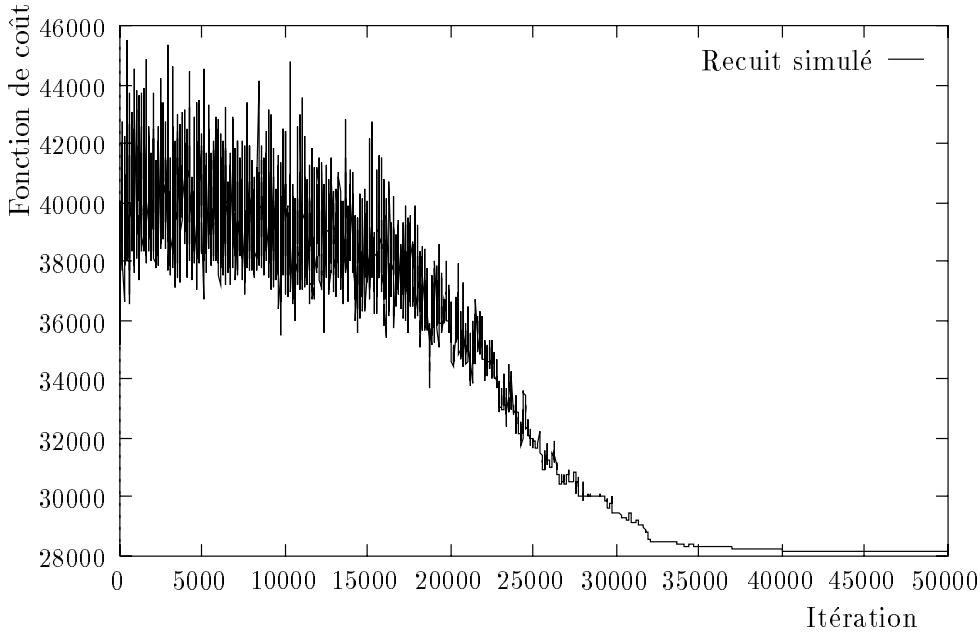


FIG. 5.7 - : Recuit simulé

Recherche Tabu

Voici les différents points caractérisant une recherche tabu (cf section 4.4.2) et les implémentations et paramétrages réalisés afin de l'adapter pour le problème du placement [GL92][BBTW94] :

- Deux types de voisinage sont utilisés : le transfert d'une tâche du processeur le plus chargé (calcul et communication) et l'échange d'une tâche située sur le processeur le plus chargé avec une autre tâche communiquant avec celle-ci et située sur un autre processeur. Cette fonction de voisinage en rapport direct avec la fonction objective permet de se promener lentement le long de celle-ci (cf figure 5.8).

- Chaque élément de la liste tabu contient une description du déplacement d'une tâche vers un processeur. Afin de diminuer les possibilités de voisinage, une liste tabu ne contenant que le déplacement de poids de calcul des tâches est utilisée. Ceci correspond au fait de considérer que dans un programme parallèle, il existe souvent un grand nombre de tâches identiques, instances d'un même modèle.

- La taille de la liste tabu a été fixée empiriquement en vue d'éviter des bouclages dans la recherche d'une bonne solution. La figure 5.9 illustre la recherche d'une solution avec une liste tabu de 10 éléments qui boucle au bout d'un certain nombre d'itérations (aux environs de 40) et le comportement du même algorithme avec une liste tabu de 100 : à ce moment là, il ne boucle pas, parvient à sortir de l'optimum local pour trouver une meilleure solution.La recherche tabu implémentée a une taille de liste tabu en relation avec le nombres de groupes de tâches.

- Comme critère d'aspiration, on considère une amélioration de la meilleur valeur rencontrée de la fonction de coût.

- Trois critères d'arrêt sont utilisés :

  - Quand tous les voisins correspondent aux éléments de la liste tabu et que le critère d'aspiration ne peut être mis en œuvre.
  - Au bout d'un certain nombre d'itérations sans améliorer la meilleure valeur rencontrée de la fonction de coût.
  - L'utilisateur peut fournir à l'algorithme un temps maximum disponible. Au bout de cet temps, la recherche tabu est interrompue et la meilleure solution trouvée est fournie.

La partie "liste tabu de taille 100" de la figure 5.9 est simplement un agrandissement de la figure 5.8. On peut remarquer dans le graphique 5.8 que la recherche tabu améliore fortement la solution lors des premières itérations. Par la suite, vers l'itération 35, des chemins détériorant (temporairement) la solution obtenue sont explorés. On remarque dans la figure 5.8 que la fonction de coût continue longtemps à être améliorée de manière laborieuse. Le recuit simulé de la figure 5.7 est tracé à partir du même problème. On voit que les deux manières de procéder sont assez différentes. Remarquons que le grand nombre d'itérations du recuit n'implique pas que celui-ci est plus coûteux. En effet une itération dans le recuit représente l'exploration d'un voisin quelconque de la solution actuelle ; tandis que dans la recherche tabu, une itération implique l'exploration de tous les voisins de la solution actuelle afin de trouver le meilleur de ceux-ci (compte tenu de la liste tabu et du critère d'aspiration).

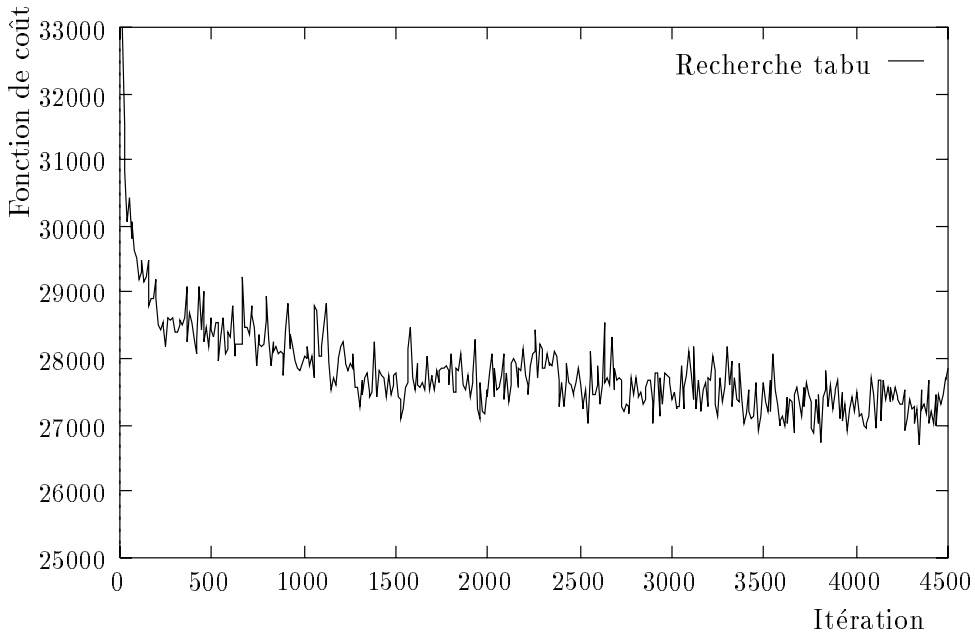


FIG. 5.8 - : Evolution de la fonction de coût pour une recherche tabu

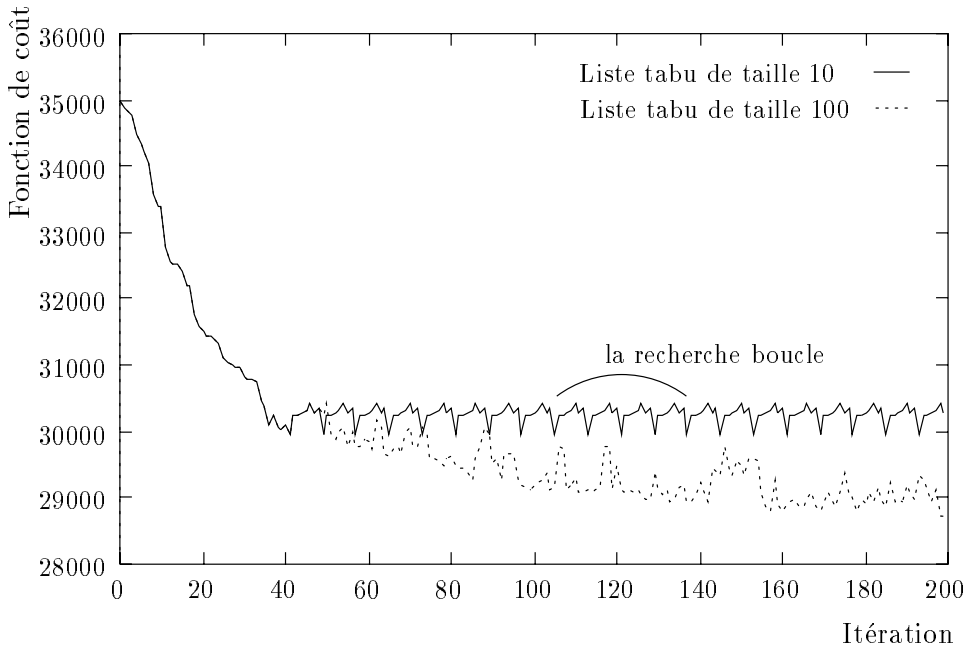


FIG. 5.9 - : Comparaison de deux tailles de liste tabu

5.6.3 Les algorithmes exacts

Meilleur d'abord et A*

L'algorithme A* est un algorithme de recherche basé sur une exploration d'arbre où les nœuds ayant les plus grandes "chances" de mener à la meilleure solution sont développés en premier. Ces chances sont estimées grâce à une partie exacte, la partie de l'arbre déjà développée et grâce à une heuristique. Il consiste à placer progressivement les tâches sur les processeurs en explorant un arbre de recherche décrivant toutes les combinaisons possibles.

L'arbre de recherche est formé de la manière suivante : On choisit, de manière arbitraire, une tâche que l'on place successivement sur les $|P|$ processeurs. Toutes les configurations correspondant au placement d'une autre tâche à partir de ces solutions partielles sont alors générées et ainsi de suite jusqu'à ce que toutes les tâches soient placées. La profondeur de l'arbre correspond au nombre de tâches.

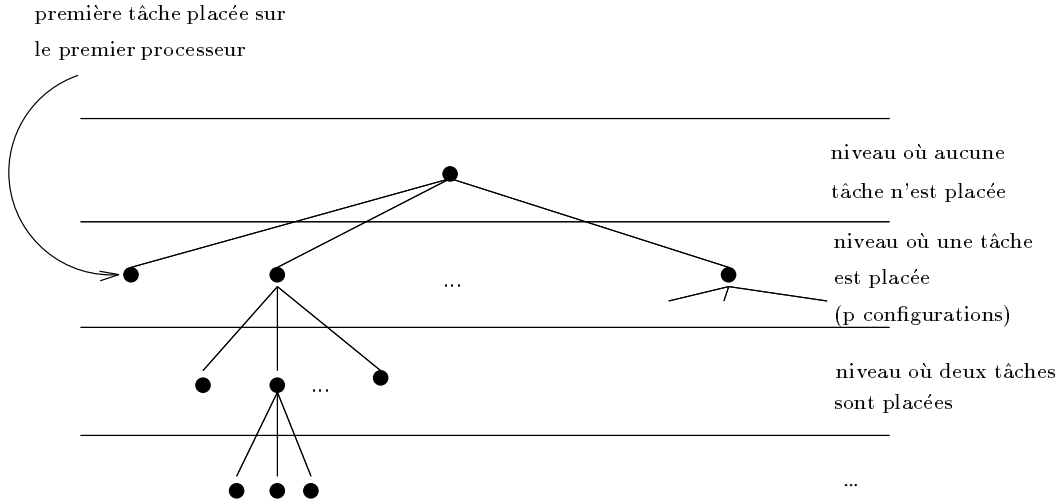


FIG. 5.10 - : Arbre de recherche du A*

Chaque nœud n de l'arbre de recherche correspond à une configuration partielle, on calcule le coût g(n) du placement partiel obtenu auquel on ajoute une estimation h(n) du coût du meilleur placement des tâches restantes. Cette somme f(n) = g(n)+h(n) est une sous-estimation du coût des placements obtenus à partir du placement partiel réalisé au nœud n. Dans l'arbre d'évaluation, on développe simultanément tous les nœuds, en commençant par ceux pour lesquels la valeur f(n) est la plus petite. Dès qu'on obtient des placements complets (feuilles de l'arbre de recherche), il est possible de couper toutes les branches pour lesquelles la sous-évaluation est plus grande que le coût du meilleur placement complet obtenu. L'efficacité de cet algorithme dépend étroitement de la finesse de la fonction de sous-évaluation h. Cette fonction est appelée fonction heuristique du A*.

La qualité du placement est estimée grâce à la fonction globale de coût. L'heuristique choisie est la suivante :
Après le placement de la i^{eme} tâche, une sous-estimation du coût de placement de la $i+1^{eme}$ tâche est calculée $(min(calc(i+1), comm(i, i+1)))$, ceci permettant de choisir pour cette tâche le même processeur que pour la i^{eme} ou un autre. Si i est sur le processeur le plus chargé (le processeur déterminant pour la fonction de coût), alors on prend cette sous-estimation pour $h(n)$, sinon on prend zéro.

5.7 Solution proposée

La boîte à outils de placement qui a été développée fait partie d'un environnement de programmation parallèle complet. Si les couches supérieures (il peut s'agir ici de l'utilisateur), fournissent un graphe orienté, un regroupement est effectué à l'aide de l'algorithme DSC de Pyrros (cf fig 5.11) ; s'il s'agit d'un graphe non-orienté, il peut être directement traité par la boîte à outils.

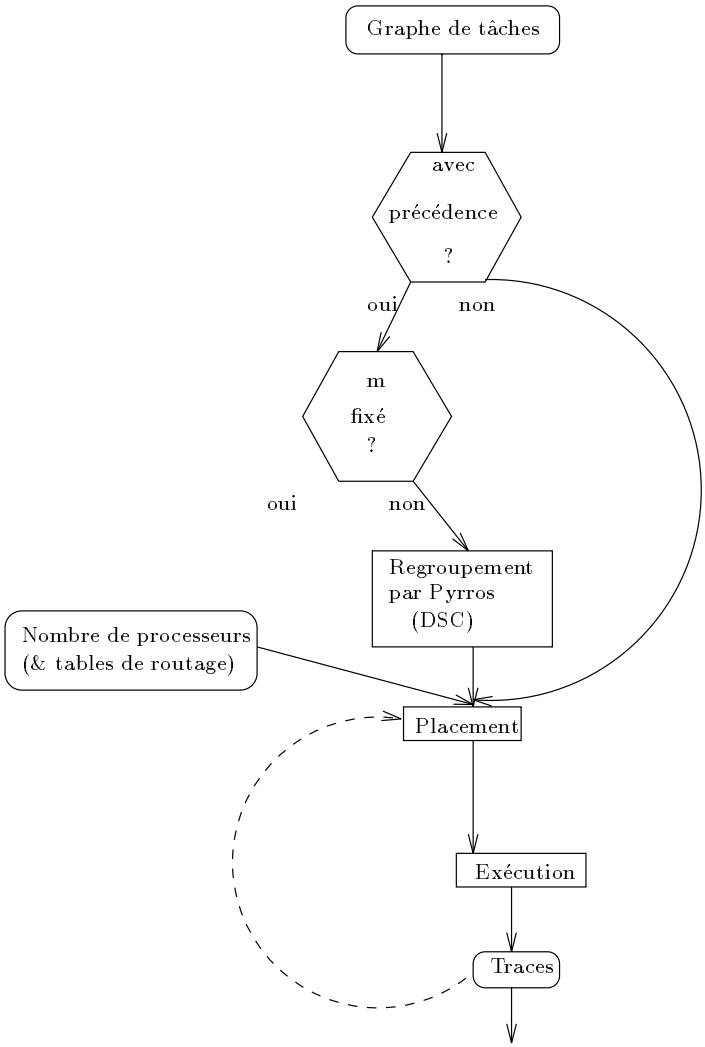


FIG. 5.11 - : Intégration de l'outil de placement et de Pyrros.

Trois types de programmes parallèles peuvent être différenciés au niveau du placement :

1. Si le programme ne comporte que peu de tâches (≤ 16) et est destiné à être exécuté un grand nombre de fois, un algorithme exact de type A* peut être utilisé. Cette solution prend beaucoup de temps, mais fournit le meilleur placement possible.

2. Si le temps passé à effectuer le placement ne peut pas être trop important et/ou que le nombre de tâches est très grand, un algorithme glouton s'impose.

3. Dans le cas général, la solution suivante est proposée (cf schéma 5.12) :

Un premier placement est effectué à l'aide d'un glouton. Si certains paramètres sont inconnus, des valeurs sont prises par défaut. Ensuite le programme est exécuté sur la machine parallèle avec une série d'outils de prise d'information (*monitoring*). Finalement l'analyse des mesures fournit des informations plus précises sur le comportement du programme. Ces informations sont utilisées dans un algorithme itératif (de type recuit ou tabu) afin d'améliorer le placement obtenu. Ces différentes phases peuvent être itérées jusqu'au moment où un placement satisfaisant est obtenu.

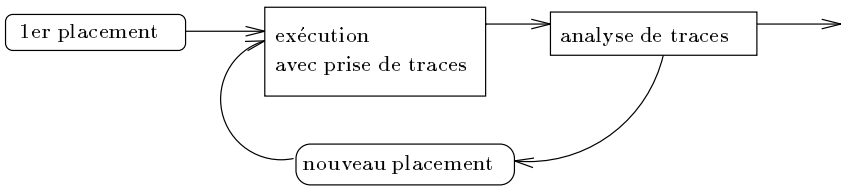


FIG. 5.12 - : Amélioration du placement après exécution

Une deuxième utilisation des algorithmes de placement, est de fournir la liste des algorithmes disponibles à l'utilisateur et de laisser celui-ci faire son choix à l'aide d'indications de performances espérées ou obtenues.

L'intégration de tels outils à un environnement de programmation demande un ensemble d'outils compatibles, ce qui va amener une définition des sorties minimales des outils de prise de traces, du générateur de graphes sans précédences (Pyrros ou l'utilisateur, ou l'analyse de traces). Une série de filtres devra aussi être mise en place pour permettre le passage d'un format de données à un autre.

Afin de faciliter la validation d'un placement, les outils de placement doivent aussi être capables de fournir une série d'indications sur la qualité du placement qu'ils ont fourni (par la suite il faudra voir si cette qualité correspond à la réalité).

5.7.1 Intégration à un environnement de programmation

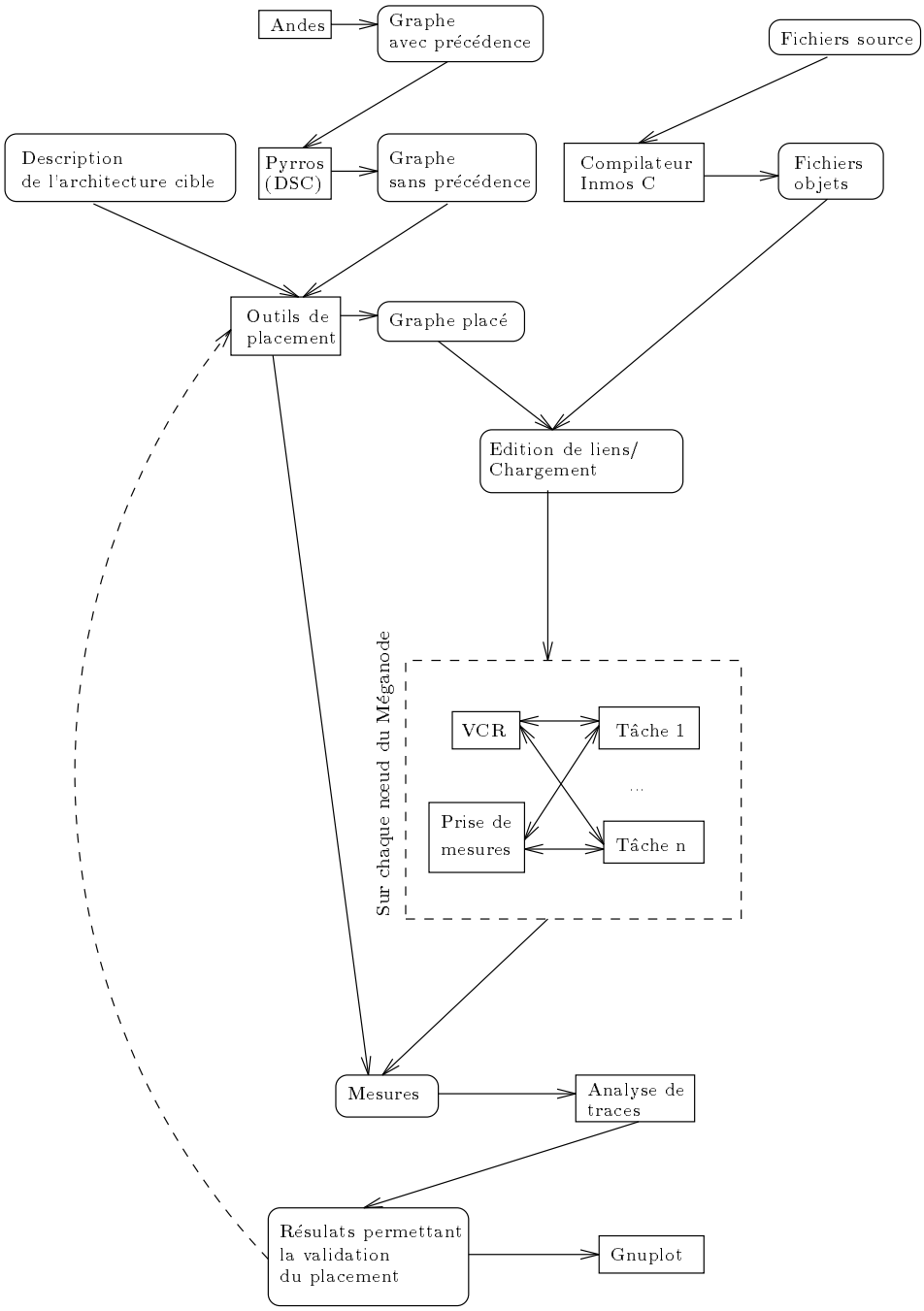


FIG. 5.13 - : Intégration de l'outil de placement sur le Méganode

Le schéma 5.13 représente l'intégration de l'outil de placement dans l'environnement tournant sur le Méganode dans le cas où Andes et Pyrros sont utilisés.

Andes est l'outil de génération de programmes synthétiques de l'environnement d'évaluation de performances ALPES qui nous a aidé à valider notre approche et qui sera décrit plus particulièrement dans le chapitre suivant.

Le graphe de tâches peut être issu de Pyrros mais peut aussi être fourni directement par l'utilisateur. Pour ce faire un langage de description de graphes a été conçu et l'analyse de ce langage utilise les outils de traitement de compilation standard que sont YACC et LEX.

Des informations concernant le placement sont fournies par les outils d'analyse de traces.

La description de l'architecture cible peut comprendre soit simplement le nombre de processeurs dans le cas d'une architecture complètement connectée soit le nombre de processeurs et les tables de routage.

5.7.2 Description des outils de prise de traces

Les outils de prise de traces nous donnent les informations suivantes:

- le temps total d'exécution du programme,
- le nombre d'octets communiqués entre chaque couple de tâches,
- le coût de calcul de chaque tâche,
- le temps d'inactivité de chaque tâche en attente de communication,
- le nombre d'octets communiqués entre chaque couple de processeurs,
- le temps d'inactivité de chaque processeur (non-compris celui succédant à la fin d'exécution de toutes les tâches placées sur le processeur).

L'analyse de ces données devrait aider le placement en déterminant de bonnes valeurs moyennes des coûts de communication et de calcul et en détectant certains problèmes locaux tels que les problèmes de congestion.

5.8 Prise en compte de contraintes supplémentaires

Les transputers du Méganode possèdent des mémoires de capacité assez restreinte (la plupart sont équipés de seulement un Méga-octet) et la plupart des programme s'exécutant sur cette machine ne pourraient donc pas se retrouver sur un nombre plus petit de processeurs. Les algorithmes de placement ont tous été adaptés afin de tenir compte de cette limitation. Le langage de description de graphes (cf annexe C) a été conçu afin de pouvoir préciser la taille mémoire

des processeurs et celle prise par les tâches. Les algorithmes gloutons veillent à ce que les tâches placées le soient sur un processeur qui ait la place pour les accueillir. De même, les relations de voisinage de l'algorithme du recuit simulé et de la recherche tabu sont conçues de manière à ne permettre que les échanges possibles au niveau mémoire.
Des indications permettant le regroupement forcé de certaines tâches sur un même processeur peuvent être fournies aux algorithmes via le langage de description de graphes. En effet, certaines tâches doivent être regroupées sur un même processeur afin d'effectuer, par exemple, des accès à une mémoire partagée. Après un regroupement effectué par PYRROS, les groupes formés sont simplement décrits comme un ensemble de tâches destinées à être sur un même processeur.

5.9 Conclusion

Une méthodologie de résolution du problème du placement a été proposée. Cette méthode dépend du graphe de tâches à placer et de son utilisation. Elle s'appuie sur une série d'algorithmes de différentes complexités et performances. Entre autres, un recuit simulé, un algorithme de recherche tabu et un algorithme de type Branch&Bound (A*) ont été spécialement adaptés pour le problème du placement. Cet outil a été complètement interfacé avec un environnement de programmation : lien avec les outils de prise de traces, lien avec un langage de description de graphes, lien avec le routeur VCR (analyse des tables de routages) et finalement lien avec les outils d'évaluation de performance qui vont nous permettre dans le chapitre suivant de valider notre approche.

Chapitre 6

Validation

6.1 Introduction

Les outils de relevé de traces permettront dans ce chapitre de valider les différentes fonctions de coût et les différents algorithmes de placement. Un jeu de tests est défini, décrit et utilisé. Les tests sont effectués sur le méganode (machine à 128 transputers), en utilisant comme routeur VCR de Southampton. Sont utilisés également les outils de génération de graphes synthétiques ANDES du projet ALPES [KP92], l'algorithme de regroupement DSC de PYRROS[YG92]. Andes, un outil de génération de graphes synthétiques (représentatifs d'une famille de programmes), a été utilisé afin de permettre de nombreuses expérimentations sur machine réelle. Andes a été développé par Joao-Paulo Kitajima au sein de l'équipe d'évaluation de performances de Brigitte Plateau (LGI-IMAG). La partie validation pratique des algorithmes de placement a été réalisée en étroite collaboration entre nos deux équipes et est présentée en tant qu'exemple d'utilisation de Andes dans la thèse de Joao-Paulo Kitajima [Kit94].

6.2 Graphes aléatoires

Afin de paramétrer les différents algorithmes de placement, des tests ont été faits sur des graphes générés aléatoirement. Les paramètres de génération sont les suivants :

- le nombre de tâches,
- le coût maximum de calcul d'une tâche,
- le coût maximum d'une communication,
- le degré maximum d'une tâche (le nombre de tâches vers laquelle elle communique),

Les différents graphes sont générés de manière uniforme en tenant compte des paramètres. Le tableau ci-dessous représente des valeurs moyennes (chaque valeur est fournie pour une centaine de graphes) d'amélioration en pourcentage apportées par les différents algorithmes par rapport à un comportement moyen de l'algorithme modulo (20000 modulo pour chaque ligne). Les paramètres correspondant au tableau ci-dessous sont : 100 tâches, 16 processeurs, un coût de calcul maximum de 1000 et un coût total de communication par tâche de maximum 1000*r (cf tableau pour les différentes valeurs de r qui est le rapport communication/calcul) et une tâche communique au maximum à 4 autres tâches. Les résultats obtenus l'ont été avec la fonction de coût utilisée, SUM, prend en compte la somme des calculs et communications et ne considère pas le routage (i.e., l'ensemble des processeurs sont à même distance).

Amélioration de la fonction en % par rapport à l'algorithme modulo					
r	lpt	lgcf	struc_lgcf	recuit simulé	recherche tabu
0.01	29.04	29.09	20.00	29.87	29.70
0.1	25.75	27.80	19.76	29.70	29.04
0.5	14.53	24.02	19.25	32.05	32.11
1	7.15	23.09	21.12	35.17	37.37
2	3.78	26.87	26.04	41.29	45.72
3	1.89	28.97	27.01	43.51	49.16
4	1.16	30.28	29.13	45.69	51.73

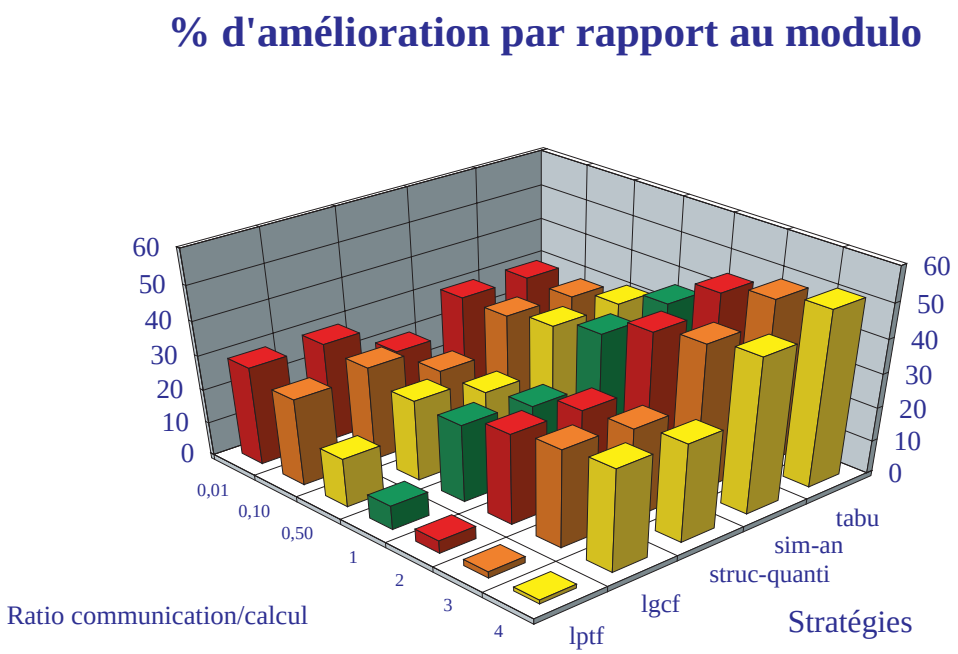


FIG. 6.1 - : Comparaison sur graphes aléatoires.

Ces tests (cf figure 6.1) correspondent exactement à notre attente :

- Les algorithmes les plus sophistiqués sont les plus performants.
- L'algorithme LPTF ne se comporte bien qu'avec peu de communications.
- Plus le rapport communication/calcul augmente, plus les algorithmes itératifs se montrent intéressants.

Dans le tableau suivant, la perte d'optimalité (en pourcentage par rapport à une recherche exacte) des différents algorithmes est décrite pour : 4 tâches, 2 processeurs, un coût de calcul de maximum 100, un coût de communication de maximum 50 et un degré maximal de sortie de 3.

Détérioration par rapport à l'optimal						
	modulo	lpt	lgcf	struc_lgcf	recuit simulé	recherche tabu
r=0.5	-46.26	-33.32	-19.36	-21.60	-8.14	-8.66

Les informations fournies par ces résultats ne sont pas suffisantes. En effet qui pourrait dire que les graphes générés aléatoirement sont représentatifs des programmes parallèles existants et comment montrer avec ces graphes l'adéquation

avec la réalité des fonctions de coût utilisées? Une approche différente s'impose : essayer de trouver des tests représentatifs des programmes parallèles et effectuer des tests sur machine parallèle réelle.

6.3 ANDES: Un outil d'évaluation de performances

6.3.1 L'approche graphes synthétiques

Les modèles d'algorithmes parallèles peuvent être utilisés dans différents contextes d'évaluation de performances : la modélisation analytique, la simulation et les mesures obtenues à partir d'un système réel [Kob78], [Jai91]. Dans ce cadre, les modèles des algorithmes font partie de la modélisation de la **charge** imposée à un modèle d'ordinateur parallèle (ou à l'ordinateur parallèle réel).

Dans une **modélisation analytique**, le langage d'abstraction est le langage mathématique. Pour les modèles analytiques en général, différentes théories mathématiques sont employées : la théorie de probabilités et des processus stochastiques, la théorie des files d'attente, les réseaux de Petri et les réseaux d'automates. On obtient les indices de performances en résolvant les modèles, soit par des techniques algébriques, soit par des techniques numériques. La description du modèle en soi n'est pas coûteuse, mais sa résolution peut l'être. Plus le modèle est proche de la réalité, plus la modélisation est complexe et plus les indices obtenus sont précis.

La **simulation** consiste à reproduire le comportement du modèle de la machine parallèle à l'aide d'événements provenant de mesures sur une charge réelle (i.e., de vrais programmes) ou modélisée (i.e., de modèles de programmes). Les contraintes imposées sur cette "re-exécution" définissent la précision de la simulation. En général, la simulation apporte des résultats plus complets que la modélisation analytique, mais remarquons que ces résultats sont aussi moins facilement extrapolables. Un autre aspect important est la possibilité de contrôler l'exécution. Par exemple, puisque le temps est modélisé, on peut arrêter la simulation et revenir dans un état antérieur (revenir dans le temps). La collecte de traces de simulation se fait aussi sans perturber l'ordre naturel des événements. D'autre part, les simulations sont faites par logiciel et peuvent être très coûteuses en temps et en espace mémoire.

La troisième approche considère que la machine parallèle est disponible et que l'on peut y effectuer des **mesures**. La charge peut être réelle ou synthétique (voir ci-dessous la définition d'une charge synthétique). A partir de l'exécution des programmes parallèles (réels ou synthétiques), on fait des mesures du système

en utilisant des moniteurs (logiciel, matériel ou hybride). Les moniteurs fournissent des traces qui sont traitées afin de donner des indices de performances. Cette technique impose l'existence de la machine parallèle, mais, en revanche, les indices calculés sont plus proches de la réalité (de la machine parallèle) que ceux obtenus par les modèles analytiques et par la simulation. Un atout de l'exécution sur une vraie machine parallèle est que les traces obtenues tiennent compte de la surcharge (en anglais *overhead*) imposée soit par la machine, soit par le système d'exploitation. En général, cette surcharge est difficile à modéliser de façon analytique ou dans une simulation. Cette technique implique une surcharge supplémentaire générée par la présence du moniteur des traces. C'est du domaine de la recherche de produire des moniteurs qui perturbent le moins possible ou qui permettent d'obtenir une bonne estimation de cette intrusion, afin de pouvoir la décompter lors de l'analyse des traces.

Nous avons choisi la troisième approche afin de modéliser toutes les surcharges liées au parallélisme et aussi parce que nous avons un multiprocesseur à mémoire distribuée disponible dans notre environnement. Nous voulons aussi évaluer des algorithmes par des modèles et non faire des expériences avec des programmes réels. En effet, nous avons décidé d'utiliser des programmes synthétiques, car les programmes réels sont difficiles à écrire. Un programme parallèle synthétique [Pop90] est un vrai programme. Cependant, un programme synthétique ne résout pas de problème. Dans son "corps" il n'y a pas de vrais calculs ou de vraies entrées/sorties. Les calculs sont des boucles vides. Par exemple, considérons un programme écrit en C qui fait l'inversion d'une chaîne de caractères (programme extrait de [KR88]) :

```
/* Ce programme accepte une chaine de caracteres comme entree et
   produit la chaine inversee. Par exemple, si l'entree est
   "abc", la sortie sera "cba" */

/* "string.h" contient la declaration de "strlen"  */
/* "strlen (c)" calcule la longueur de la chaine c */
#include <string.h>

/* inverse la chaine s */
main()
  {
    int c,        /* lettre intermediaire de la chaine */
        i, j;     /* utilises dans les iterations      */
    char s[30];   /* chaine acceptee et inversee       */

    scanf ("%30c", s);         /* on lit la chaine */

    /* inversion de la chaine */
    for (i = 0, j = strlen(s)-1; i < j; i++, j--)
        {
```

```
            c = s[i];       /* echange de lettres */
            s[i] = s[j];
            s[j] = c;
        }

    printf ("%30s\n", s);  /* affichage de la chaine inversee */
  }
```

Un programme synthétique possible pour le programme présenté cidessus est :

```
main()
  {
     int i;

     /* instruction synthetique qui simule la fonction "scanf",
        c'est-a-dire, la lecture de la chaine */
     for (i = 0; i < SCANF_TIME (STRING_SIZE); i++);

     /* instruction synthetique qui simule l'iteration */
     for (i = 0; i < (STRING_SIZE / 2); i ++);

     /* instruction synthetique qui simule la fonction "printf",
        c'est-a-dire, l'affichage de la chaine inversee */
     for (i = 0; i < PRINTF_TIME (STRING_SIZE); i++);
  }
```

Les fonctions `SCANF_TIME` et `PRINTF_TIME` expriment la quantité d'opérations équivalentes aux opérations d'entrée et de sortie respectivement. Ces valeurs peuvent être obtenues, par exemple, à partir d'une prise de mesures lors de tests. `SCANF_TIME` et `PRINTF_TIME` comporte un paramètre :`STRING_SIZE`. `STRING_SIZE` peut représenter la longueur de la chaîne lue, si on sait d'avance cette valeur, ou une valeur moyenne.

Notre approche consiste à générer des programmes synthétiques à partir d'un modèle *ANDES-C* et à partir d'un modèle de machines. Un modèle *ANDES-C* est composé de paramètres qui peuvent être modifiés afin de modéliser différents programmes parallèles. Le changement des paramètres d'un modèle *ANDES-C* peut être aussi dirigé par un modèle de machines : par exemple, un coût de calcul plus grand modélise une machine avec des processeurs plus lents. L'interaction entre le modèle de programme et le modèle de machines, la génération rapide de programmes synthétiques et la possibilité de tester plusieurs stratégies d'implémentation représentent le cœur de l'outil *ANDES-Synth*.

6.3.2 Le principe de *ANDES-Synth*

L'environnement *ANDES-Synth* est un outil développé au sein du projet *ALPES* (*AL*gorithmes, *P*arallélisme et *E*valuation de *S*ystèmes)[KP92][BKPT94].

L'objectif de cet outil est de permettre l'évaluation des performances de systèmes parallèles par des **exécutions d'une charge synthétique sur une machine parallèle réelle**.

La structure de base de *ANDES-Synth* est présentée dans la Figure 6.2. Cette structure se compose :

1. d'un modèle quantitatif de l'application parallèle (la charge de travail) écrit en *ANDES-C*;
2. d'un modèle d'ordinateur parallèle;
3. de stratégies d'implémentation (e.g., algorithmes de groupement, d'ordonnancement, de placement, d'équilibrage de charge) utilisées afin de permettre l'exécution d'une charge synthétique sur la machine parallèle émulée (émulation basée sur les informations contenues dans le modèle de machines).

A partir de l'exécution réelle de la charge synthétique, des traces sont obtenues. L'analyse des performances est faite sur ces traces.

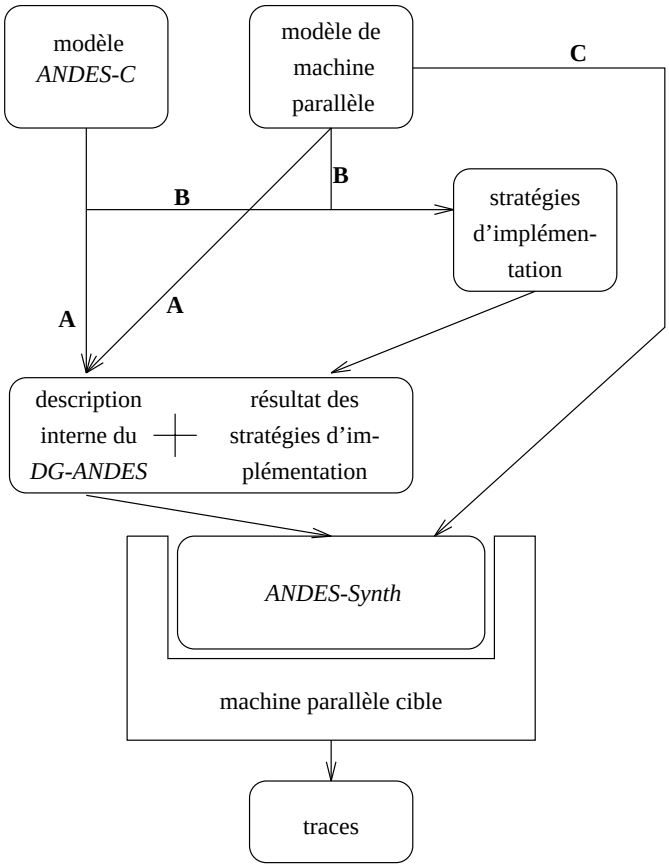


FIG. 6.2 - : L'environnement *ANDES-Synth*.

6.3.3 Les objectifs

Notre objectif est de comparer les performances de différentes stratégies de placement statique avec l'outil *ANDES-Synth* (c'est-à-dire, avec des modèles de programmes parallèles et des modèles de machines parallèles – la technique d'évaluation est l'exécution de charges synthétiques). Le critère de comparaison est le **temps d'exécution** de la charge synthétique correspondante à un modèle de programme et de machine données.

La qualité des stratégies de placement est comparée par rapport à un indice de performance (le temps d'exécution) obtenu à partir de l'exécution d'une charge synthétique sur une machine parallèle **réelle**. La durée d'une exécution réelle peut être divisée en trois parties : le temps d'exécution de tâches, le temps de surcharge de la machine (e.g., la communication et le système d'exploitation) et le temps d'inactivité. Cette durée d'exécution de la charge synthétique doit être proche du temps d'exécution de l'application parallèle réelle, modélisée par cette charge synthétique.

En opposition à ce temps précis, les fonctions de coût des stratégies de placement donnent un temps **approximatif** d'exécution d'un programme parallèle sur une machine parallèle (les deux représentés par leurs modèles respectifs). De cette façon, plus que comparer les temps d'exécution donnés par l'outil *ANDES-Synth*, notre objectif est aussi de vérifier si les coûts d'exécution de la réalité sont conformes aux coûts d'exécution théoriques (donnés par les fonctions de coût des algorithmes de placement). Si ce n'est pas le cas, il faut essayer d'expliquer les raisons de la disparité des performances.

6.3.4 Le jeu de tests

Il n'existe pas un jeu de tests d'algorithmes parallèles parfaitement représentatif qui puisse être utilisé dans nos expériences. La représentativité est importante pour la généralisation des conclusions tirées à partir de la comparaison des stratégies de placement. Nous avons alors développé un jeu de tests de 17 modèles d'algorithmes. Ces modèles sont exprimés en *ANDES-C* et ils sont dérivés de la littérature (principalement des livres d'algorithmique parallèle, par exemple, [BT89]) et de *GENESIS 2.2* [Hey91], un jeu de tests de programmes parallèles pour les multiprocesseurs à mémoire distribuée et à échange de messages. Nous espérons que la représentativité de notre tests est adéquate. Les modèles présentés de manière détaillée en annexe sont donnés à *ANDES-Synth* pour l'évaluation des stratégies de placement. Avant de parler de modèle de machine, nous présentons une caractérisation de notre batterie de tests basé sur certains paramètres quantitatifs et structuraux que nous jugeons important dans un contexte d'évaluation des performances. Un aspect essentiel dans la construction du jeu de tests est la normalisation des tailles de problème entre les différents modèles. Nous avons alors supposé que tous les problèmes traitent une matrice

de 32x32 éléments. Certains problèmes comme `BELLFORD2` et `PROLOG` ne traitent pas explicitement ce type de structures de données. Pour `BELLFORD2`, le graphe sur lequel les chemins les plus courts doivent être calculés peut être représenté par une matrice de connexion. Pour `PROLOG`, on suppose que la recherche d'un but à partir des faits (dans ce travail, un programme logique est défini par un but à atteindre à partir d'un ensemble de faits) nous amène à un ensemble maximal de 32x32 possibilités. D'autres problèmes qui traitent de matrices avec 32x32 éléments ont une quantité de nœuds de calcul qui ne peuvent pas être gérés par l'outil *ANDES-Synth*. Dans ce cas, on diminue proportionnellement la taille du graphe, mais on augmente les coûts de calcul de chaque nœud.

Voisinage structurel. Le nombre de communications par nœud de calcul est un critère structurel utilisé souvent dans les stratégies d'implémentation de programmes parallèles. Plus de communications arrivent ou sortent d'un nœud de calcul, plus le programme parallèle perd du temps à les gérer (par exemple, à chaque envoi de message, un temps de démarrage de communication).

bench	iter2	prolog	stra2	divcq	gauss	diam1	warsh	fftr2	diam4
#arcs/#nœuds	2,20	3,00	3,08	3,33	3,87	3,87	3,93	4,00	5,38
bench	diam3	msgas	diam2	ms3	bell2	pde01	pde02	qcd02	
#arcs/#nœuds	5,40	5,64	5,75	5,90	7,11	8,97	8,99	24,49	

Parallélisme virtuel. On peut estimer le nombre moyen de nœuds de calcul qui s'exécutent simultanément en divisant le nombre de nœuds de calcul par le nombre maximal d'étages du *DG-ANDES*.

bench	ms3	qcd02	gauss	diam1	diam2	diam4	msgas	bell2	warsh
nœuds	3,72	15,55	15,59	15,78	15,78	20,88	25,71	28,08	30,18
bench	diam3	prolg	divcq	stra2	iter2	pde02	pde01	fftr2	
nœuds	33,40	77,08	80,74	92,69	213,50	216,77	366,00	426,83	

Les paramètres ci-dessus sont structuraux et ils ne changent pas même si les caractéristiques quantitatives du *DG-ANDES* (e.g., nombre total d'opérations) changent. Ces caractéristiques quantitatives (exprimées en fonction du **nombre** d'opérations et d'octets) doivent être transformées afin qu'elles puissent être comparées. Les coûts de calcul et de communication sont alors transformés en **temps** estimés de calcul et de communication. Néanmoins, pour calculer ces temps, il faut connaître le modèle de machines associé.

6.3.5 Les modèles de machines

Le modèle de machines choisi pour nos expériences représente un ordinateur très proche du multiprocesseur cible sur lequel *ANDES-Synth* s'exécute (le Méganode). Le modèle de machines représente un ordinateur parallèle à mémoire

distribuée composé de 16 processeurs de calcul (tore 4x4) et de 2 processeurs auxiliaires (un processeur interface avec l'hôte et un processeur utilisé pour la fermeture du tore). La raison de ce choix est tout d'abord liée à la structure du Méganode. Une telle topologie est construite en utilisant seulement un module du Méganode, où les temps de communication entre les processeurs ne sont pas variables pour une taille de message donnée. Cela nous donne un modèle de communication linéaire simple et fiable pour une distance donnée et une longueur de message donnée (un *packet*). Le tore a été choisi à cause de son faible diamètre.

Les ressources de calcul. Nous considérons un processeur de base comme étant un Transputer qui ne réalise que des opérations entières. A partir de mesures faites sur un Transputer réel, le modèle opérations (n_{ops}) versus temps de calcul (t_{calc}) employé est :

$$t_{calc} = 8.71 + (1.76 * n_{ops})$$

Le processeur de base est multiprogrammé et on peut régler son degré de multiprogrammation. Dans nos expériences, nous disposons d'un modèle de machines sans multiprogrammation (degré 1 de multiprogrammation) et sans préemption : lorsqu'un nœud de calcul s'exécute, il le fait jusqu'à sa fin en rendant pas la main. Nous avons fait des expériences en relaxant ces contraintes : avec des degrés 5 et 10 de multiprogrammation et un modèle avec préemption où, à chaque 1000 itérations vides, le processus de calcul se met à la fin de la file de processus actifs. Finalement, les nombre entiers sont représentés par 4 octets dans cette architecture.

Les ressources de communication. L'échange de messages dans le modèle a le même comportement que l'échange de messages dans le Méganode avec un routage logiciel par paquets (de 160 octets) envoyés en pipeline. Les communications ne sont pas totalement synchrones (le rendez-vous n'est pas parfait) car le récepteur d'un message envoie un acquittement lors de la réception du premier paquet et pas lors de la réception de tout le message. De cette façon, l'émetteur peut se débloquer avant la réception complète du message par le destinataire. Le routage est statique : les messages prennent le même chemin entre deux processeurs pendant une exécution. Des modèles quantitatifs ont été développés pour exprimer le temps de communication (t_{comm}) en fonction de la taille du message (n_{octets}). Pour une communication entre deux processeurs voisins, on a :

$$t_{comm} = 188.38 + (0.98 * n_{octets}) \quad n_{octets} \leq 160$$

$$t_{comm} = 210.03 + (1.49 * n_{octets}) \quad n_{octets} > 160$$

Pour une communication à distance 2, on a :

$$t_{comm} = 297.16 + (2.16 * n_{octets}) \quad n_{octets} \leq 160$$

$$t_{comm} = 464.50 + (1.63 * n_{octets}) \quad n_{octets} > 160$$

Pour une communication à distance 3, on a :

$$t_{comm} = 358.92 + (3.22 * n_{octets}) \quad n_{octets} \leq 160$$

$$t_{comm} = 711.55 + (1.61 * n_{octets}) \quad n_{octets} > 160$$

Enfin, pour une communication à distance 4, on a :

$$t_{comm} = 438.64 + (4.37 * n_{octets}) \quad n_{octets} \leq 160$$

$$t_{comm} = 999.19 + (1.67 * n_{octets}) \quad n_{octets} > 160$$

Ces modèles ont été obtenus à partir des mesures réalisées sur le Méganode, configuré en tore 4x4, sans l'existence d'une charge d'interférence causée par d'autres communications. Des modèles plus fins (où la distance est donnée comme paramètre et où une charge moyenne du réseau est considérée) sont présentés dans [TP94]. Dans l'avenir, ces modèles feront partie de nos modèles de machines.

Les ressources de mémoire. Ces ressources n'ont pas été prises en compte pour les expériences réalisées avec *ANDES-Synth*. Néanmoins, les stratégies de placement ont besoin d'une quantification de ces ressources, afin de ne pas placer plus de tâches que la mémoire d'un processeur puisse contenir. Actuellement, les modèles de machines donnés au placement possèdent des processeurs avec une taille illimitée de mémoire.

Retournons aux caractéristiques de notre jeu de tests avec la connaissance du modèle de machines présenté ci-dessus.
Dans le tableau ci-dessous se trouvent décrit le rapport communication/calcul, les coûts moyen de calcul ($E(calcul)$), les écarts-type de calcul (σ_{calcul}), les coûts moyen de communication ($E(communication)$) et les écarts-type des communications des graphes groupés par PYRROS ($\sigma_{communication}$).

	% de comm	$E(calcul)$	σ_{calcul}	$E(comm)$	σ_{comm}
warshall	20.52	125997.26	57803.74	65078.35	55916.49
bellford2	5.53	142334.38	64708.87	16674.13	8252.34
diamond1	2.06	583661.61	107689.54	24576.00	0.00
diamond2	50.13	291979.74	102991.07	587060.90	8534.84
diamond3	3.82	178487.58	46403.62	14166.91	3559.69
diamond4	18.97	47270.82	34538.92	22136.47	11502.31
divconq	2.68	52939.51	26835.83	2920.16	4040.37
pde1	33.06	82797.58	31707.35	81783.11	64126.74
fft2	12.82	55511.41	30154.16	16326.68	41044.74
gauss	5.94	301501.61	163499.36	38080.00	12471.79
iterative2	17.06	44357.76	3892.48	18244.53	145676.99
ms3	5.20	1762751.77	7657594.26	193370.87	845148.65
ms_gauss	5.54	976333.04	760348.90	114413.23	77382.87
pde2	23.02	119748.02	60139.90	71617.78	50365.20
prolog	5.95	35411.36	25897.42	4480.00	25466.84
qcd2	6.68	12056936.40	3003856.45	1724800.00	425098.65
strassen2	12.58	15128.39	13779.89	4352.48	6381.36

6.3.6 Tests effectués

L'ensemble du jeu de tests (17) a été exécuté sur le Méganode avec pour chaque test, 3 degrés de multiprogrammation différents (nombre de processus utilisateurs tournant en même temps sur un processeur), avec 3 rapports communication/calcul différents, les 4 fonctions de coûts et en étant groupé de 2 manières différentes (par Pyrros et manuellement). Tout cela pour 6 à 9 algorithmes de placement différent (3 algorithmes ont été abandonné pour manque de performances), ce qui représente donc une dizaine de milliers d'expériences sur machine parallèle.
Parmi les données récoltées, on retrouve des données issues du placement : le temps pris par le placement, la valeur de la fonction de coût, le temps total de calcul, le temps total d'exécution, le placement obtenu. On retrouve aussi une série de données issues des outils d'évaluation de performances : le temps d'exécution de l'algorithme placé, les informations sur les graphes de précédence, les temps d'activité (système et utilisateur) et d'inactivité des processeurs, le temps de calcul effectivement passé et le temps de communication.

6.4 Analyse des résultats

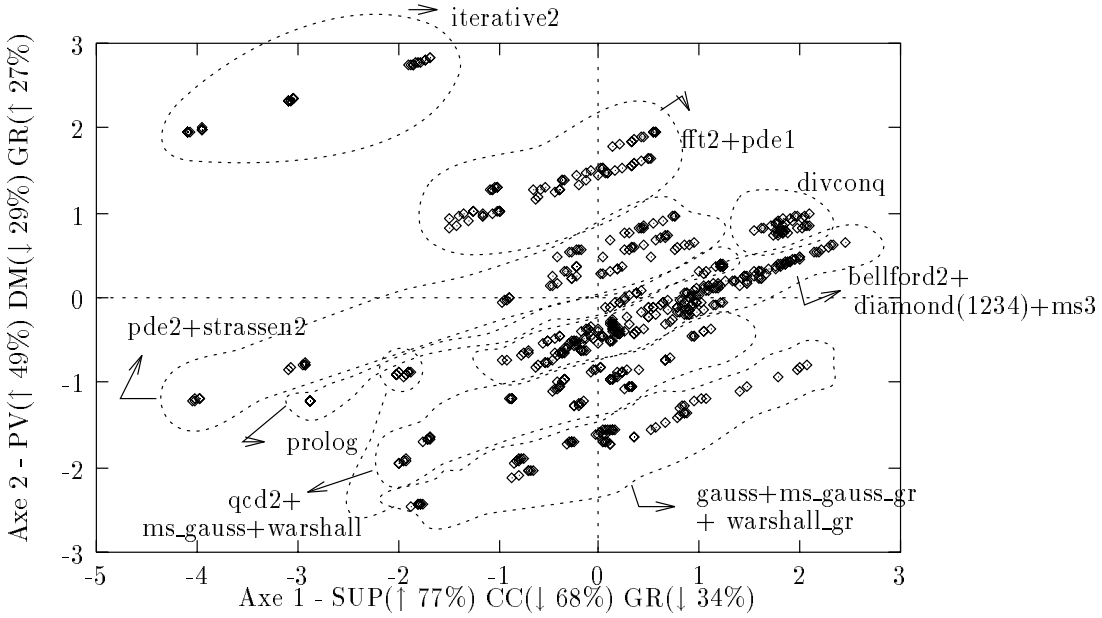
6.4.1 Représentativité des tests

Les caractéristiques des tests effectués (très structurés et peu de communications) correspondent bien aux panel des programmes parallèles existants. En effet, suite aux recherches effectuées, nous avons trouvé cinq types de programmes :

- Des programmes issus de la programmation SIMD qui ont donc tendance à être émulé en plaçant le même code sur les différents processeurs.
- Des programmes Prolog qui sont sous formes d'arbres et donc très structurés.
- Des programmes crées par des scientifiques issus d'autres disciplines que l'informatique (physiciens, etc.). Ces programmes cherchent à exploiter le plus facilement possible la puissance des machines parallèles et donnent des programmes aux structures assez simples telles que par exemple les maîtres/esclaves. Généralement ces algorithmes sont à gros grains (par exemple dans la méthode de Monte-Carlo un germe est envoyé à un travailleur et ce germe de petite taille correspond à une forte charge de travail).
- Des programmes créés par des mathématiciens/informaticiens qui cherchent à exploiter de manière très fine le caractère itératif de certains algorithmes tels que FFT, GAUSS, etc. Ces méthodes donnent aussi des algorithmes très structurés, de plus ces développeurs cherchent à minimiser les temps de communications qu'ils savent coûteux.
- Des programmes créés par des mathématiciens/informaticiens qui cherchent à exploiter de manière la plus complète possible les possibilités des architectures parallèles. Ces programmes sont à grains fins, utilisent la création dynamique de processus et sont très difficilement représentables statiquement.

Il existe donc deux classes structurelles de programmes parallèles : des programmes très structurés statiques et des programmes très peu structurés dynamiques. Les algorithmes de placement proposés seraient plus avantageux à utiliser avec des programmes peu structurés et statiques. Cette catégorie de programme n'est pas courante de nos jours. Mais les développements sur machines MIMD à mémoire partagée ne font que commencer...
Nous avons réalisé une série d'ACP (Analyses en Compososantes Principales [Ber94]) afin de différencier les différents tests. Le principe de base d'une ACP est d'obtenir une représentation approchée de l'échantillon de n individus (dans ce cas, par exemple, l'ensemble des tests du *benchmark*) dans un sous-espace de dimension faible. Le critère de choix de l'espace de projection est la minimisation

de la déformation des distances (entre individus) en projection. C'est à l'utilisateur de déterminer la dimension du sous-espace en question, en fonction de ce qu'il désire et de certains critères [Ber94].
Le graphique 6.3 illustre une ACP représentant l'ensemble des tests dans un plan dont les abscisses sont composées de SUP (*Speed-Up* est l'accélération amenée par rapport à une version séquentielle du programme), du rapport communication/calcul GR (grain des tâches groupées, c'est-à-dire l'écart type de l'occurence des communications dans les tâches) et en abscisse PV (parallélisme virtuel, c'est-à-dire une approximation de la largeur du graphe), DM (le degré de multiprogrammation, c'est-à-dire le nombre de pouvant être exécutée en même temps en pseudo-parallélisme sur un processeur) et GR.



(suffixe -gr : groupement à la main)

FIG. 6.3 - : ACP : Représentation des individus (tests)

L'ensemble des tests est bien dispersé dans l'espace (cf figure 6.3), ce qui illustre la représentativité du jeu de tests pour les critères choisis. Un ensemble de groupes de tests peut être discerné dans ce graphique, ce qui est représenté par un ensemble de patatoïdes étiqueté par les individus concernés.

6.4.2 Adéquation de la fonction objective

La relation existant entre fonction de coût et temps d'exécution va nous permettre d'estimer l'adéquation de celle-ci avec la réalité. En effet les différentes fonctions de coûts tentent d'estimer le temps d'exécution. Cette comparaison va

nous permettre de voir si les simplifications (volontaires ou non) qui ont été utilisées dans les modèles de machines et de programme ne sont pas trop primordiales. Le modèle de machines visé par ce travail était un réseau complètement connecté de processeurs et le modèle utilisé est un réseau point-à-point de transputers. Certaines modifications des algorithmes ont été effectuées afin de prendre ce fait en compte (fonctions de coût $ROUT$ et TOR). Les performances obtenues sur réseau de transputers peuvent donc être considérées comme une sous-estimation des performances obtenues avec une machine correspondant plus au modèle utilisé.
Parmi ces différentes simplifications nous retrouvons :

- Le fait de considérer des coûts moyen de calcul et de communication par tâche.
- Le fait de considérer la charge du processeur plus chargé comme une estimation du temps d'exécution (pas de prise en compte des temps d'inactivité des processeurs (attentes de communications, etc.) ni de toute la charge amenée par le système).
- La part de la charge amenée par des communications est difficile à cerner. En effet, de nombreux facteurs rentrent en compte tels que :
  - Toutes les unités d'un transputer (unité de virgules flottantes, unité d'entiers, liens de communication) sont capables de fonctionner en parallèle.
  - Le routeur VCR utilise le processeur d'entiers. De plus VCR utilise une *packetisation* des messages de plus de 160 octets et pipeline ces *packets* jusqu'au processeur destination.
  - Les processeurs se trouvant sur le chemin d'une communication devant avoir lieu doivent s'occuper de la gestion de cette communication.
- De nombreux facteurs inattendus viennent perturber le comportement des programmes. Par exemple :
  - la taille mémoire des transputers (1 méga-octet) ne permet pas de lancer trop de processus en pseudo-parallélisme. Une vingtaine de processus utilisateurs tournant en pseudo-parallélisme par processeurs est un maximum de fait.
  - le système de pseudo-parallélisme utilisé sur les transputers est très limité. Il existe deux priorités, haute et basse. En haute priorité les processus ne peuvent être interrompus et en basse priorité, ils ne peuvent l'être que lors de la rencontre d'une intruction *déschédulante* (principalement un saut ou une communication).

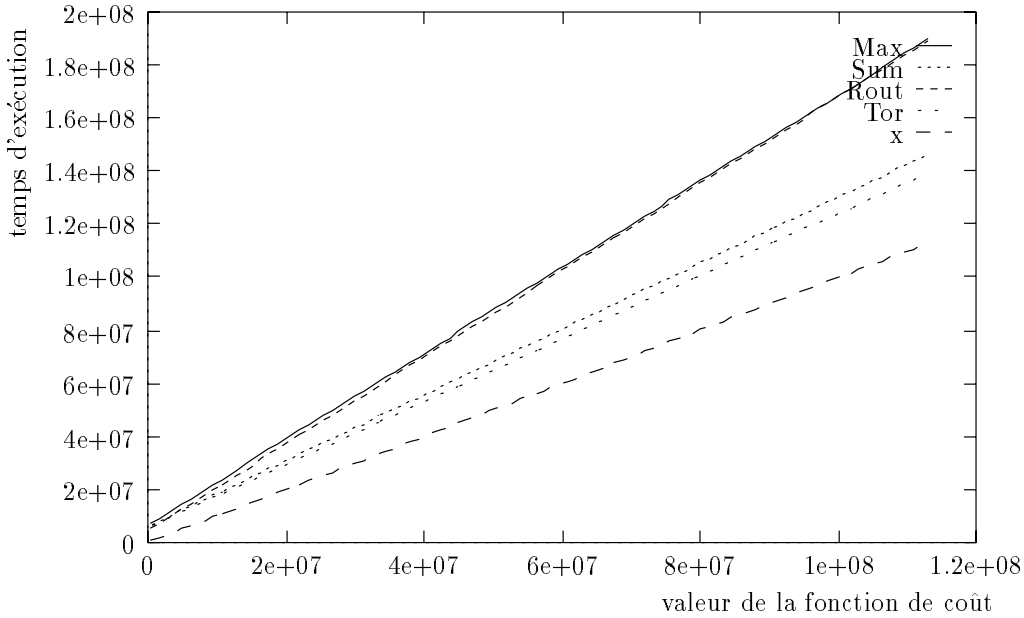


FIG. 6.4 - : Fonctions de coût et temps d'exécution

Voici les points observables dans le graphique 6.4, où sont représentés les droites de régression pour les différentes fonctions objectives (tous les couples (fonction de coût, temps d'exécution) issus des tests ont été pris en compte) :

- Il existe bien une corrélation linéaire entre fonction de coût et temps d'exécution. Le travail effectué afin de minimiser une fonction de coût n'est donc pas perdu.
- La fonction de coût qui se rapproche le plus du comportement du temps d'exécution est la fonction TOR. Son coefficient de corrélation linéaire, r^2 est de .86.
- La fonction SUM se comporte déjà assez bien. En effet le coût des distances de communication sont masqués en grande partie par le mécanisme de routage.
- La fonction MAX ne se comporte pas bien, sans doute, à cause du coût d'utilisation de VCR qui utilise le processeur de calcul.
- La fonction $ROUT$ est décevante, mais cela est explicable car le coût des communications n'est pas simplement facteur du coût de passage entre deux processeurs, du nombre d'octet et de la distance entre les deux processeurs. Cela a été montré expérimentalement grâce aux valeurs trouvées pour le tore.

- Aucune fonction n'est idéale. Cela est évident vu le manque d'information résidant dans la représentation sous forme de graphes de tâches. Les fonctions de coûts pourraient cependant encore être affinées en prenant en compte un réseau chargé, etc.

6.4.3 Amélioration apportée

Dans le tableau de comparaison des gains obtenus avec les différents algorithmes par rapport à l'algorithme modulo (cf annexe B), pour la fonction TOR et le groupement effectué par PYRROS, on peut différencier plusieurs types de résultats :

$\oplus\oplus$ De très bons résultats pour les graphes DIAMOND (1-4), PDE (1-2) et STRASSEN : de 12 à 29 % du temps d'exécution avec de nombreuses améliorations aux alentours de 17 %.

$\oplus$ De bonnes améliorations pour les graphes BELLFORD, DIVCONQ, GAUSS, PROLOG et WARSHALL : jusqu'à 13 % du temps d'exécution avec de nombreuses améliorations aux alentours de 7 %.

$\ominus$ De mauvais résultats pour les graphes ITERATIVE-2, MS3 et QCD2 : en moyenne 2 à 3 % de pertes par rapport à un modulo moyen.

Remarquons un cas où la prise en compte des communications apporte une forte perte par rapport à l'algorithmes LPTF : MS3 où le groupement effectué par PYRROS donne un groupement représentant un cas pathogène de notre modèle de prise en compte des communications.

6.4.4 Performance des différents algorithmes

L'ensemble des algorithmes gloutons a toujours donné des résultats en moins d'une seconde sur un SUN-4, tandis que les algorithmes itératifs peuvent prendre jusque plus d'une heure.
Au niveau des différences de résultats amenés par les algorithmes (cf Annexe-B), on peut distinguer :

- LPTF se dénote au niveau du comportement par rapport aux autres algorithmes.
- LPTF est mieux que Modulo dans la plupart des cas.
- La gestion des communications s'impose et un algorithme tel que LGCF a montré un excellent comportement.

- Les algorithmes itératifs n'ont pas amené de grandes améliorations au niveau du temps d'exécution, ni de la fonction de coût par rapport à l'algorithme quantitatif. Cela peut être dû à plusieurs raisons :
  - Les tests utilisés sont très structurés (i.e., possèdent un ensemble non négligeables de symmétries) et un placement de type LGCF suffit.
  - Les tests utilisés n'ont pas de grand rapport communication/calcul.

  Remarquons que dans le cas des tests le placement n'est qu'une phase et le gain gagné est à additionner au gain gagné lors du regroupement. Deux éléments nous laisse entendre que si les algorithmes itératifs n'ont pas beaucoup amélioré la solution modulo, c'est qu'il n'y avait pas beaucoup à gagner :
  - Les tests effectués sur graphes aléatoires avec un grand rapport communication/calcul montrent que des gains intéressants et que les algorithmes itératifs se distinguent bien des algorithmes gloutons.
  - Quand il y a peu de communications, on se rapproche du cas sans communication pour laquelle on connaît la borne 4/3 du LPTF. Il n'y a donc dans ce cas là, pas beaucoup à gagner.

Nous voyons dans le graphique 6.5, que si ,pour les tests illustrés, le rapport communication/calcul avait été plus élevé, les gains au niveau de la fonction de coût de l'algorithme tabu par rapport à l'algorithme LGCF auraient été plus marqués. En effet dans ce graphique, les abscisses correspondent à x, un facteur multiplicateur du rapport communication/calcul par rapport au rapport de base fourni par les algorithmes classiques. Au plus on augmente ce rapport, au plus les résultats, en termes de fonction de coût, de l'algorithme tabu sont différenciés (i.e. on gagne plus) par rapport à l'algorithme LGCF.

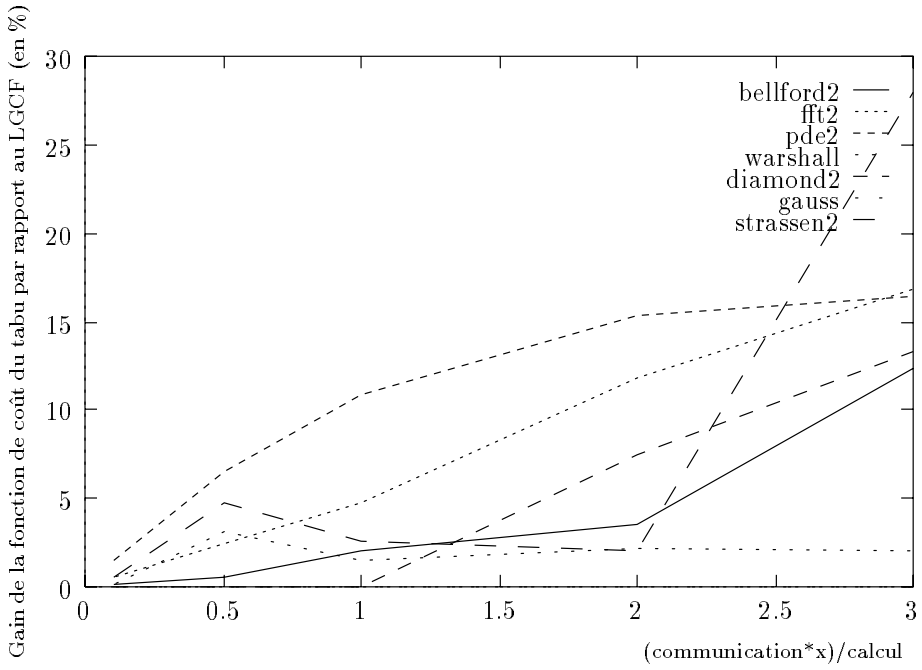


FIG. 6.5 - : Comparaison entre tabu et LGCF

6.5 Itération

Les connaissances des coûts de communication et de calcul dans nos tests étaient assez précises. Pour simuler le comportement d'un utilisateur n'ayant qu'une idée imprécise des coûts des communications et permettre une amélioration itérative (après relevé de traces), nous avons fait varier les informations passées à l'algorithme de recherche sur le graphe *diamond*2. L'algorithme exécuté garde cependant un rapport communication/calcul normal. Dans le graphique 6.6, nous voyons que plus les informations passées à l'algorithme de placement sont correctes (proches de 1), plus le temps d'exécution diminue. Ceci montre la faisabilité d'une amélioration progressive d'un placement obtenu en ayant de meilleurs indications sur les différents coûts.

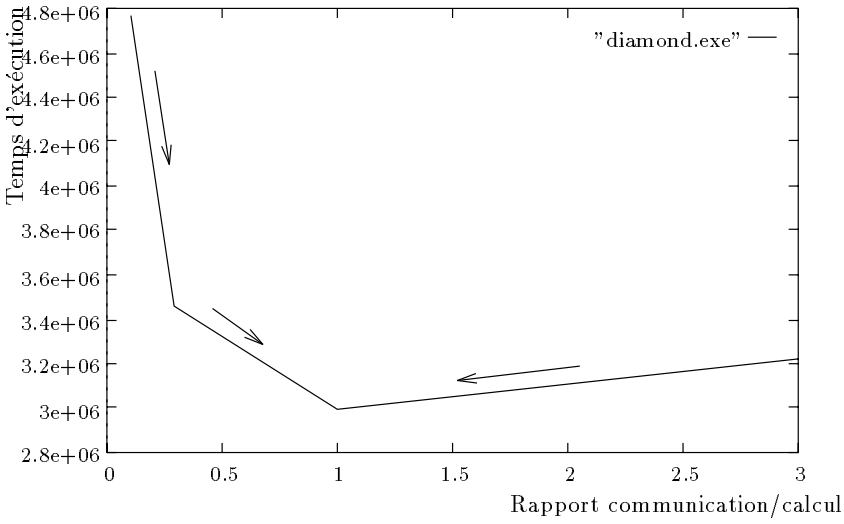


FIG. 6.6 - : Améliorations successives d'un placement

6.6 Conclusion

Un jeu de tests a été conçu et utilisé afin de montrer l'adéquation des différents éléments de la boîte à outils de placement et la réalité : les graphes de tâches, les fonctions de coûts et les différents algorithmes de placement. L'interfaçage avec PYRROS a permis en partant d'une série de graphes de précédence de les regrouper. L'interfaçage avec ANDES a permis d'émuler la charge apportée par les différents tests sur machine réelle (Méganode) et de fournir les mesures en résultant. Des statistiques sur les expériences ont été menées afin d'amener différentes conclusions :

- l'adéquation des fonctions objectives (et donc des différents modèles utilisés) avec la réalité,
- les algorithmes prenant en compte les communications donnent de meilleurs résultats que LPTF,
- pour les tests effectués, les gains obtenus avec les itératifs n'améliorent pas fortement les gains obtenus avec LGCF,
- la possibilité d'une amélioration du placement obtenu, après relevé de traces, est possible.

Chapitre 7

Conclusion et perspectives

Dans ce chapitre, une conclusion concernant la présente thèse est fournie. Nous voyons ensuite les différentes perspectives que le présent travail a ouvertes et les améliorations possibles.

7.1 Conclusion

Le modèle de machines parallèle MIMD qui nous semble le plus prometteur est celui proposé par de nombreux constructeurs aujourd'hui : une série de processeurs homogènes connectés via un réseau multi-étages avec un coût des communications inter-processeurs ne dépendant pas de ceux-ci (tous les processeurs sont à distance constante ou sont considérés comme tels). Pour des raisons historiques, les tests ont été effectués sur réseaux de transputers et les algorithmes de placement ont été adaptés (i.e. les fonctions de coût utilisées) afin de tenir compte de la distance entre processeurs.

Les différentes composantes logicielles rentrant en jeu dans un environnement de programmation parallèle ont été cernées. Celles qui influencent le placement, principalement issues du système d'exploitation, ont été décrites. Une série de paramètres basés sur le système et influençant les spécifications de modèles de programme a été énumérée. Une description détaillée de deux modèles de programmation statique a été donnée et les autres modèles ont été passés en revue. Le modèle de graphes de tâches sans précédence a été retenu pour le présent travail.

L'intégration des outils relatifs au placement de tâches dans un environnement de programmation parallèle a été décrit. Un état de l'art des solutions données aux problèmes d'ordonnancement et plus particulièrement des algorithmes destinés et utilisables pour le problème du placement a été dressé. Une double utilisation des algorithmes de placement a été donnée : non seulement en tant que solution à part entière pour des graphes de tâches sans précédence, mais aussi comme moyen de projeter sur une architecture réelle un graphe de tâches avec relation de

précédence qui aurait été regroupé (et utilise donc autant de processeurs virtuels que désirés).

Un ensemble de solutions au problème du placement a été proposé. La méthode dépend du graphe de tâches à placer et de son utilisation. Elle s'appuie sur une série d'algorithmes de différentes complexités et performances. Entre autres, un recuit simulé, un algorithme de recherche tabu et un algorithme exact de type Branch&Bound (A*) ont été spécialement adaptés pour le problème du placement. Cet outil a été complètement interfacé avec un environnement de programmation : lien avec les outils de prise de traces, lien avec un langage de description de graphes, lien avec le routeur VCR (analyse des tables de routages) et finalement lien avec les outils d'évaluation de performance qui nous ont permis de valider notre approche. Un jeu de tests a été conçu et utilisé afin de montrer l'adéquation des différents éléments de la boîte à outils de placement et la réalité : les graphes de tâches, les fonctions de coûts et les différents algorithmes de placement. L'interfaçage avec PYRROS a permis en partant d'une série de graphes de précédence de les regrouper.L'interfaçage avec ANDES a permis d'émuler la charge apportée par les différents tests sur machine réelle (Méganode) et de fournir les mesures en résultant. Différentes statistiques sur les expériences ont été fournies afin d'amener différentes conclusions :

- L'adéquation des fonctions objectives (et donc des différents modèles utilisés) avec la réalité.
- Les algorithmes prenant en compte les communications donnent de meilleurs résultats que LPTF.
- Pour les tests effectués, les gains obtenus avec les itératifs n'améliorent pas fortement les gains obtenus avec un glouton futé tel que LGCF qui prend en compte les coûts de calcul et de communication.
- La possibilité d'une amélioration du placement obtenu, après relevé de traces, est possible.

En résumé :

- Une étude des modèles de programmation et de machines existants a été menée. La représentation sous forme de graphe de tâches des programmes parallèle a été retenue.

Le choix des modèles et l'adaptation d'une série d'algorithmes de complexités et performances différentes, nous a amené à la réalisation d'une boîte à outils de placement et à l'écriture d'une série de conseils et de méthodes d'utilisation de celle-ci.

- Cette plate-forme a été interfacée avec de nombreux outils existants (PYRROS, ANDES, etc.) et validée sur machine réelle (le Méganode).

- La validation a montré l' adéquation des méthodes utilisées avec la réalité et leurs limites.
- La faisabilité et l'intérêt d'une plate-forme de placement statique pour programmes parallèles a par là même été démontrée.
- L'intérêt de la phase de placement et donc des outils développés lors du présent travail a aussi été montré dans le cas où le modèle de graphes de précédence a été choisi et qu'un regroupement a été effectué sur une architecture virtuelle (à nombre de processeurs non borné). Le placement servant alors à se ramener à une architecture réelle.

Dans le cadre de l'informatique parallèle, la recherche de performances est primordiale. Il a été montré dans cette thèse que veiller à effectuer un bon placement statique pour un programme était une manière peu coûteuse (les algorithmes gloutons ont toujours pris moins de 1 seconde) de gagner sur le temps d'exécution. Ceci a été montré non seulement pour les programmes décrits sous forme de graphe de tâches sans précédence, mais aussi pour ceux décrits sous formes de graphes de précédence sur lesquels on effectue un regroupement.

7.2 Perspectives

Dégageons des différentes conclusions de ce travail des perspectives à court et à long terme :

Des retombées directes du présent travail sont envisageables. En effet la masse d'informations tirée des tests réalisés est telle que de nombreux résultats peuvent encore en être extraits. Une exécution des tests pour un type de groupement donné (manuel ou automatique) fournit en effet 35 méga-octets compactés d'informations à exploiter. Parmi les différents résultats que l'on peut attendre sous peu :

- des résultats au niveau du travail réalisé sur des fonctions de coût prenant en compte la charge du réseau,
- des résultats sur l'adéquation des différentes familles de programmes parallèles (et donc aussi une description détaillée de celles-ci) avec la modélisation effectuée pour le problème du placement. Comme perspectives pratiques, les différentes règles pourraient conduire à un système expert qui permettrait d'effectuer les décisions de manière automatique.

- Une série d'améliorations au niveau de la convivialité et des possibilités de l'outil de placement peuvent être apportées, telles que :
  - L'interface utilisateur (X-Window, cf annexe C) peut être développée afin de fournir des indications graphiques des performances et résultats obtenus.
  - Un interfaçage avec des outils de programmation visuelle est en cours dans le cadre du projet SEPP.
- Il n'y a pas de raisons de considérer que les programmes de demain seront parallèles et pas les environnements de programmation. Nous pouvons donc envisager de :
  - paralléliser les algorithmes itératifs afin de diminuer leur temps d'exécution,
  - paralléliser les algorithmes exacts tels que le A*, ce qui permettrait d'effectuer des placements exacts (au sens de notre modèle) et de comparer ceux-ci aux placements effectués par d'autres algorithmes.

La complémentarité avec des outils de placement dynamique a été soulignée. Cette possibilité sera utilisée dans les environnements APACHE et SEPP. En effet il est envisageable d'exploiter les données connues statiquement du programme parallèle (en dépliant, par exemple, le graphe des appels aussi loin que possible) afin d'effectuer un bon placement de base. Ensuite, dès lors que la situation évolue (par exemple beaucoup de nouveaux processus sont lancés dynamiquement), un outil de répartition de charge dynamique peut prendre la main.

Annexe A

Description des tests

Ci dessous se trouvent détaillés les différents algorithmes faisant partie de la batterie de tests.

`BELLFORD2`. L'algorithme de Bellman-Ford [BT89] résout le problème de trouver les chemins les plus courts entre tous les nœuds d'un graphe orienté pondéré et un nœud destination de ce graphe. Le nœud destination n'a pas des successeurs. Dans la figure A.1, la structure du *DG-ANDES* de `BELLFORD2` est présentée pour le graphe orienté `EXEMPLE` (la fonction des arcs pointillés dans le *DG-ANDES* est de représenter un groupement manuel).

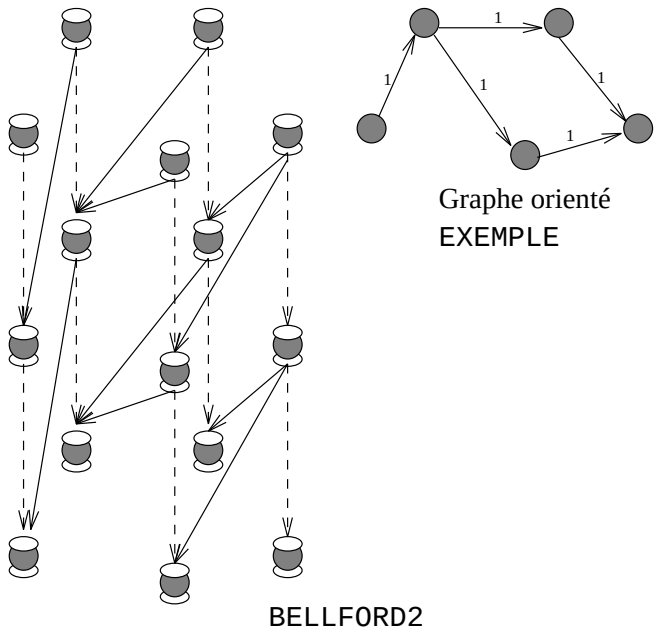


FIG. A.1 - : La structure de `BELLFORD2`.

`DIAMOND1`. Ce modèle représente un calcul systolique décrit dans [IS90]. Dans cet article, le graphe est connu sous le nom de "graphe temps-espace" (cf figure A.2).

`DIAMOND2`. Ce modèle représente un calcul systolique de multiplication de matrices décrit dans [LK90] (cf figure A.2).

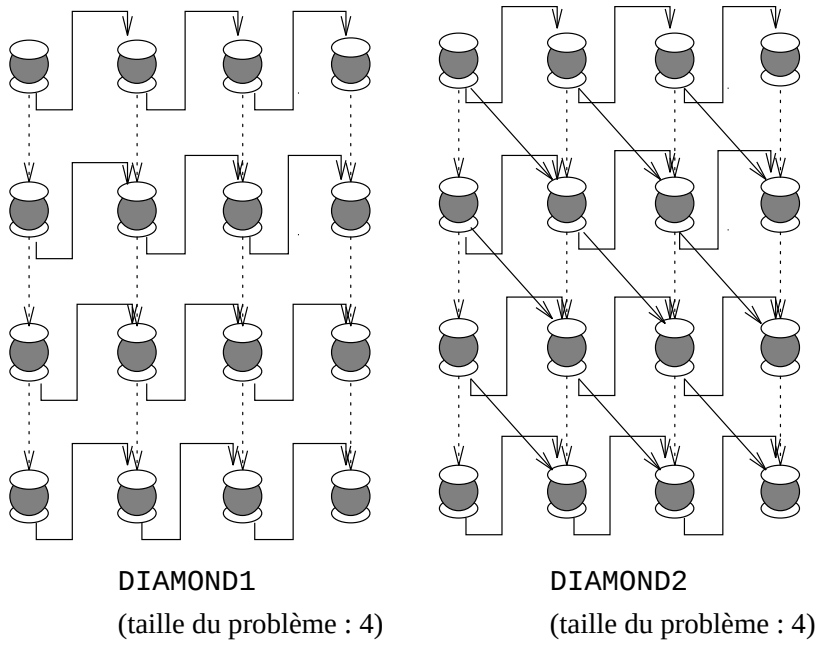


FIG. A.2 - : Les structures de `DIAMOND1` et de `DIAMOND2`.

`DIAMOND3`. Ce modèle représente un calcul systolique de multiplication de matrices décrit dans [KCN90]. Ce modèle décrit de façon plus fine la multiplication modélisée par `DIAMOND2` (cf figure A.3).

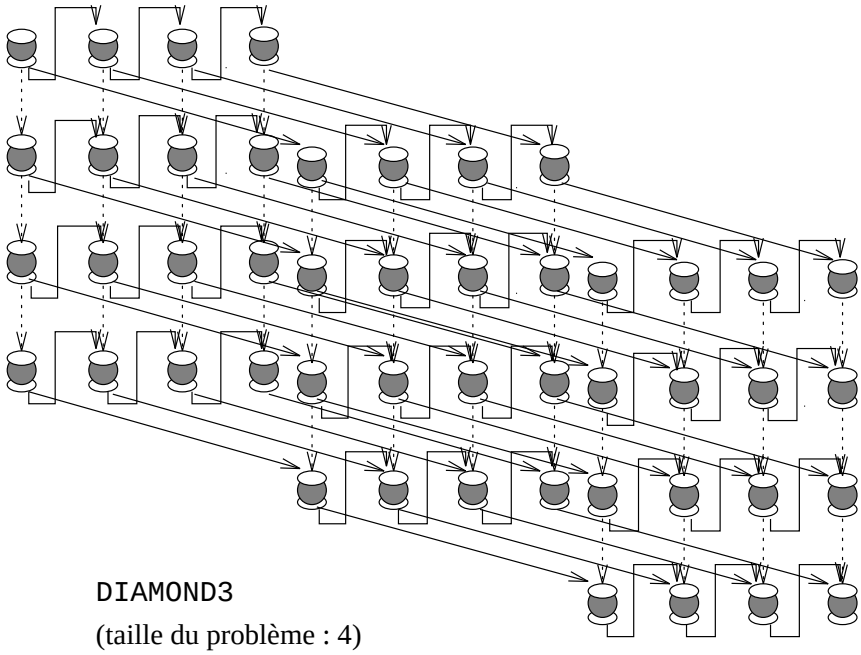


FIG. A.3 - : La structure de DIAMOND3.

DIAMOND4. Ce modèle représente un calcul systolique de la fermeture transitive d'une relation sur un ensemble d'éléments [SC92] (cf figure A.4).

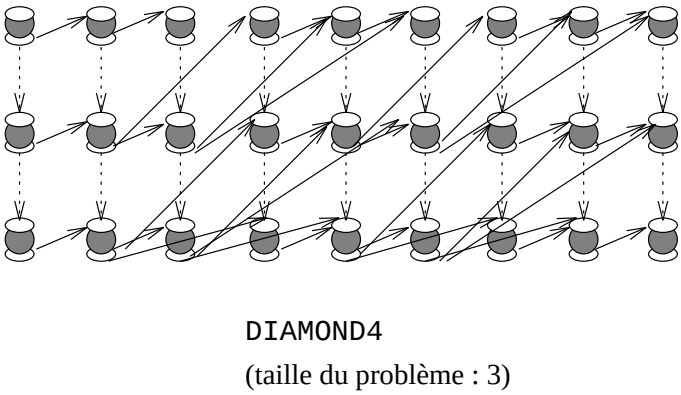


FIG. A.4 - : La structure de DIAMOND4.

DIVCONQ. Ce modèle représente un algorithme du type diviser-pour-règner [MS91] (cf figure A.5).

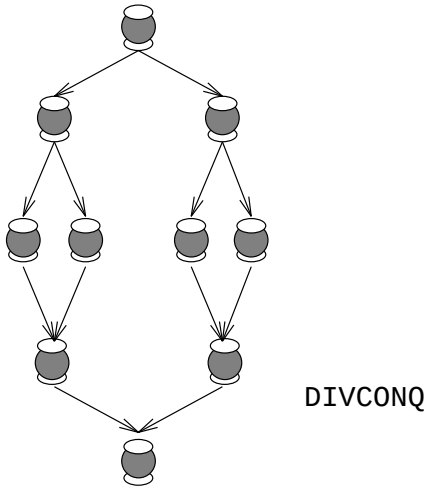


FIG. A.5 - : La structure de `DIVCONQ`.

`FFT2`. Il s'agit d'une transformée rapide de Fourier unidimensionnelle. Ce modèle a été dérivé d'un programme de *GENESIS 2.2* (cf figure A.6).

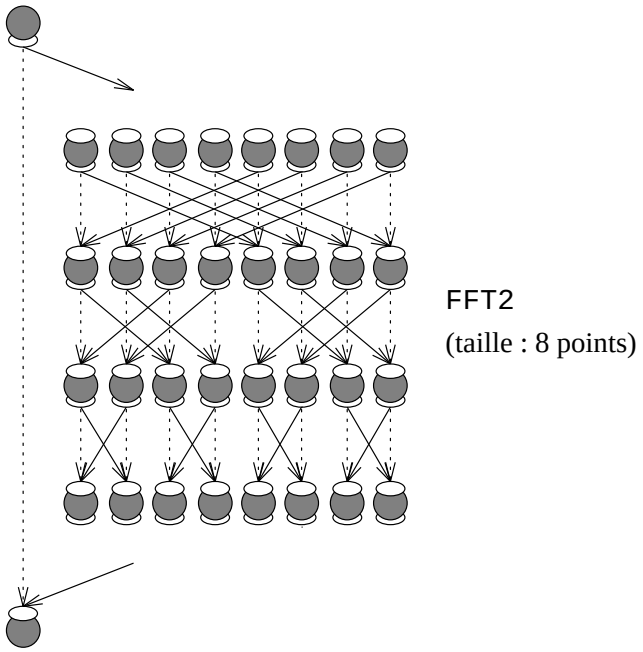


FIG. A.6 - : La structure de `FFT2`.

GAUSS. Ce modèle correspond au graphe d'exécution de l'algorithme d'élimination de Gauss utilisé dans la résolution de systèmes linéaires [BT89][CT93] (cf figure A.7.

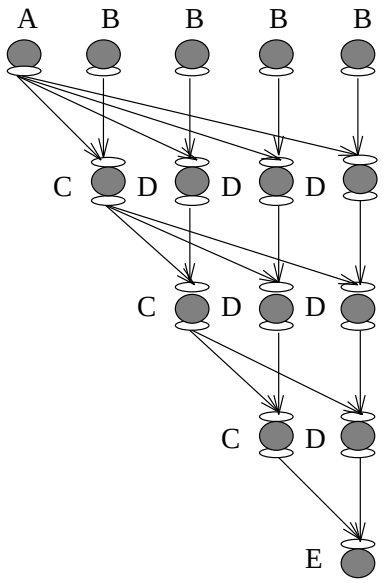


FIG. A.7 - : La structure de GAUSS

ITERATIVE2. Ce modèle représente un graphe d'exécution d'un programme itératif générique. Chaque étage correspond à une itération. A la fin de chaque itération, chaque nœud de calcul réalise une diffusion des données vers les nœuds de l'étage successeur (cf figure A.8).

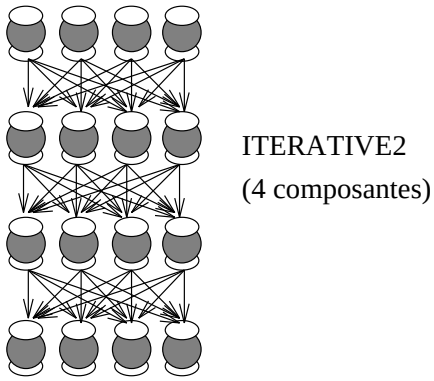


FIG. A.8 - : La structure de ITERATIVE2.

`MS3`. Ce *benchmark* correspond à un modèle générique d'un algorithme du type série-parallèle [MS91] ou maître-esclave. `MS3` est plus détaillé que le graphe présenté dans [MS91], car l'exécution des "esclaves" est décomposée en un ensemble d'itérations (igure A.9).

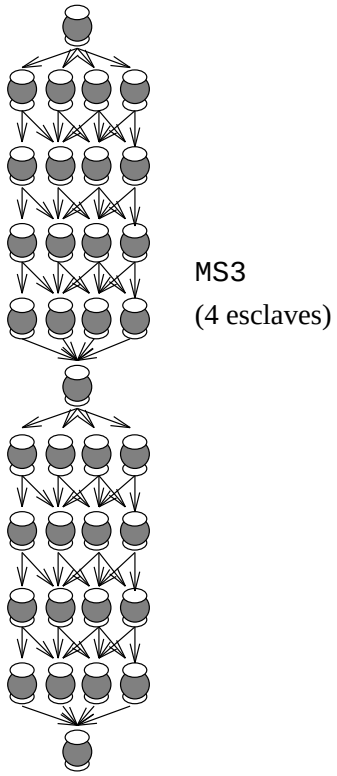


FIG. A.9 - : La structure de `MS3`.

`MS_GAUSS`. Ce modèle correspond à la concaténation du modèle du maître-esclave avec le modèle de l'élimination de Gauss. La phase "maître-esclave" représente, par exemple, un traitement sur une matrice et la phase "élimination de Gauss" modélise la résolution du système linéaire associé à la matrice traitée antérieurement.

`PDE1`. Il s'agit du modèle d'un algorithme de résolution de l'équation de Poisson dans l'espace tridimensionnel. La méthode employée est la relaxation rouge-noir. Ce modèle a été dérivé du programme `PDE1` de *GENESIS 2.2* (Figure A.10).

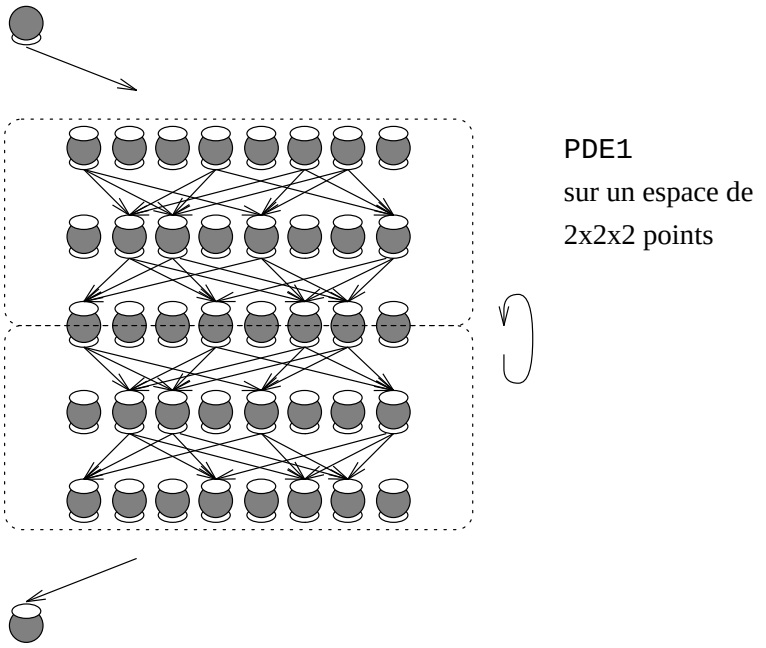


FIG. A.10 - : La structure de PDE1.

PDE2. Modèle similaire à PDE1, mais sur l'espace bidimensionnel. La méthode de résolution de l'équation de Poisson est la multi-grille (en anglais *multi-grid*). PDE2 est aussi un *benchmark* de *GENESIS 2.2* (Figure A.11).

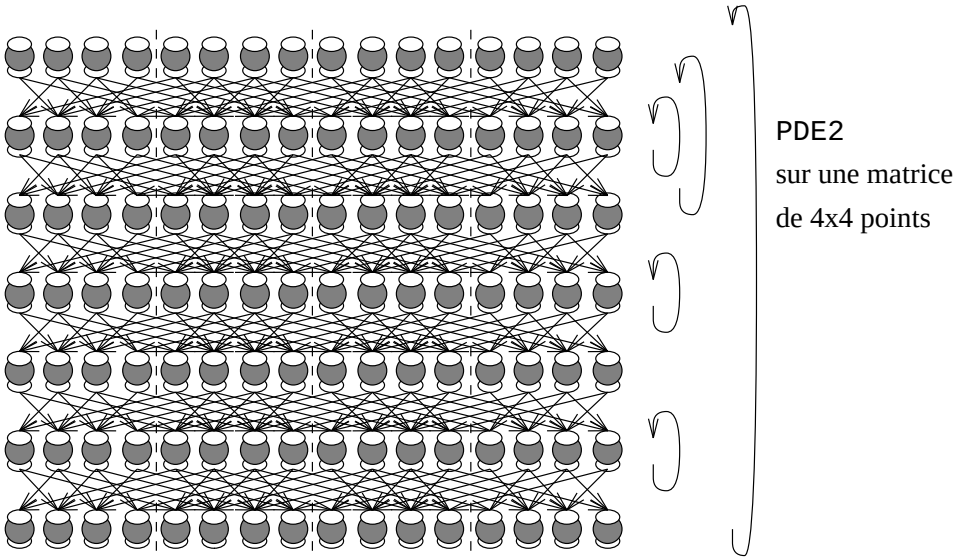


FIG. A.11 - : La structure de PDE2.

PROLOG. La précédence des activités de la machine d'inférence d'un langage PROLOG peut être modélisée par un arbre (cf figureA.12).

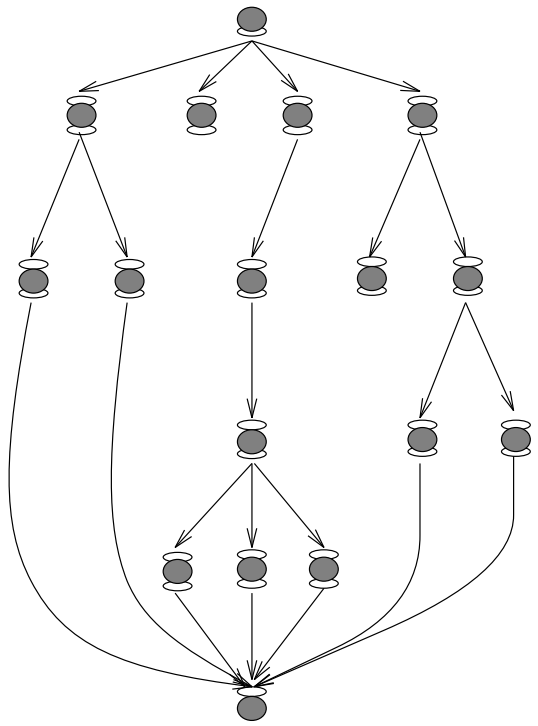


FIG. A.12 - : La structure de PROLOG.

QCD2. Basé aussi sur le *benchmark* correspondant de *GENESIS 2.2*, QCD2 est le modèle *ANDES* d'un algorithme de résolution de systèmes linéaires. La méthode employée est la gradient conjugué [BT89] (Figure A.13).

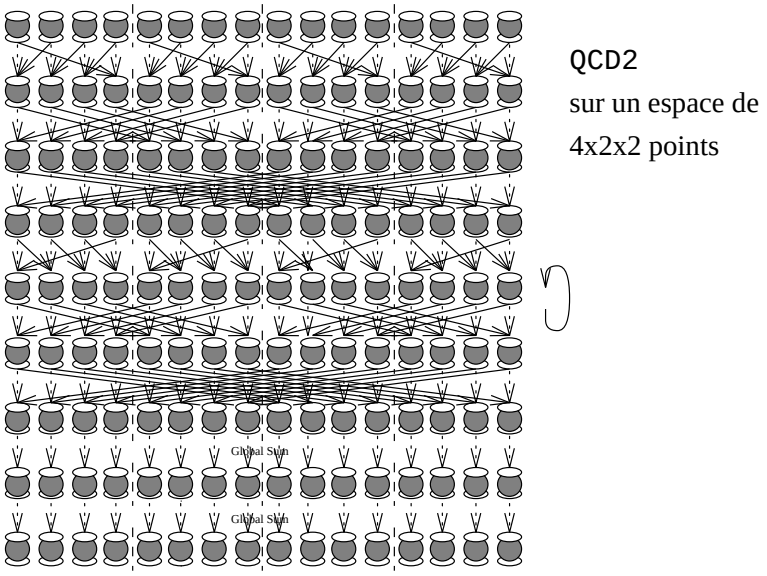


FIG. A.13 - : La structure de QCD2.

STRASSEN2. L'algorithme de Strassen [CT93] est une méthode utilisée pour la multiplication rapide de matrices. Les matrices sont décomposées en sous-matrices. Des opérations d'addition et de multiplication sur ces sous-matrices sont alors effectuées. La méthode de Strassen peut être appliquée de nouveau pour la multiplication de sous-matrices. La structure des nœuds de calcul du modèle de l'algorithme de Strassen est présentée dans la fgure A.14.

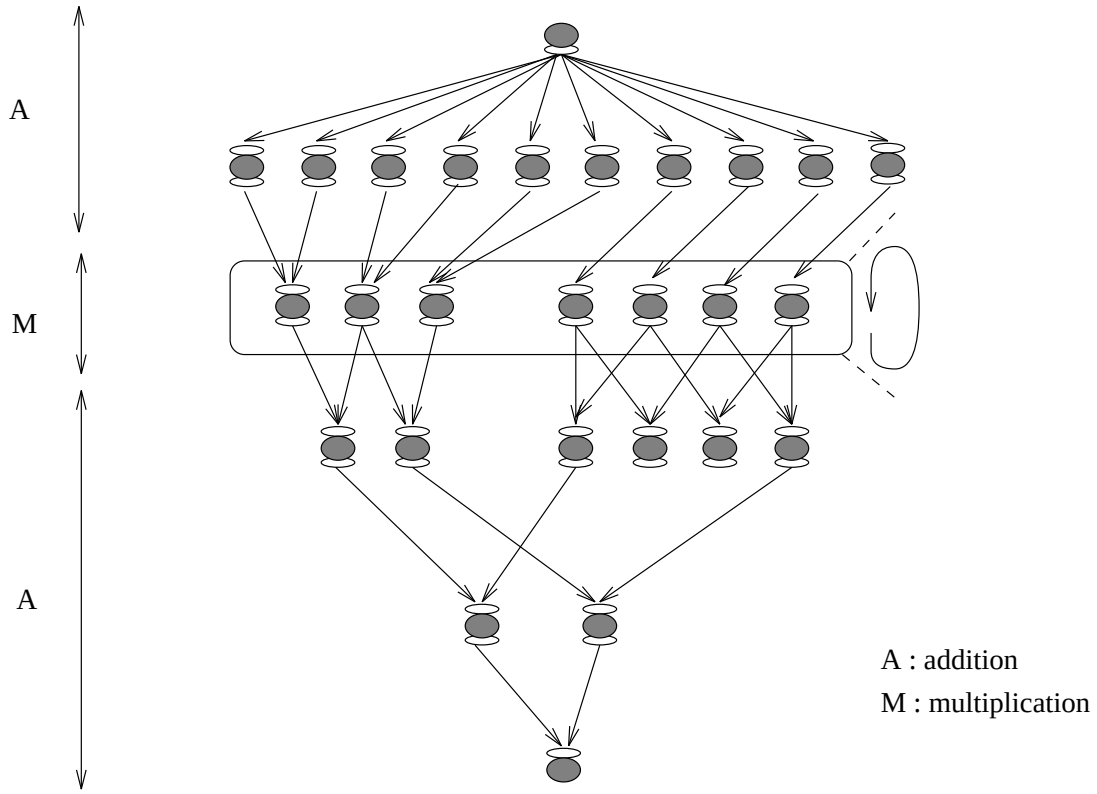
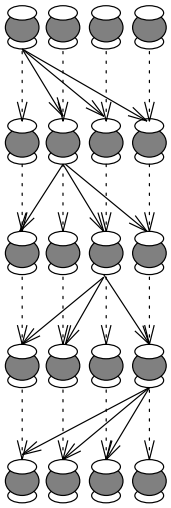


FIG. A.14 - : La structure de STRASSEN.

`WARSHALL`. Ce modèle correspond au modèle de l'algorithme de Warshall présenté dans [WB93]. L'algorithme de Warshall consiste à trouver la fermeture transitive d'un graphe orienté. Ce modèle fait partie du *benchmark* ParStone (lié au schéma de classification *BACS*). Le schéma de la structure du *DG-ANDES* est dans la Figure A.15.



WARSHALL
taille du problème :
4 noeuds

FIG. A.15 - : La structure de WARSHALL.

Annexe B

Récapitulatifs des tests

Les deux tableaux qui suivent reprennent les résultats des tests avec un degré de multi-programmation de 1. Chaque test est répété trois fois (avec des coûts de communication haut, moyen et faible).

Gain par rapport au modulo avec comme coût Somme

Strategy	LPT		Quanti		Struc		Sim-an		Tabu	
	funct	exec	funct	exec	funct	exec	funct	exec	funct	exec
bellford2	29.83	8.22	33.28	7.50	31.81	4.82	34.72	10.34	34.68	7.29
bellford2	26.02	10.55	27.46	10.85	27.46	6.52	28.13	12.89	28.13	8.30
bellford2	18.65	17.11	29.60	15.70	29.60	15.30	38.15	8.12	36.89	12.54
diamond1	6.92	8.74	6.92	7.49	6.92	7.61	6.92	11.66	6.92	9.40
diamond1	32.04	9.26	32.04	8.12	32.04	7.21	32.04	9.14	32.04	5.44
diamond1	33.13	0.76	33.13	-0.49	33.13	0.88	33.13	-1.07	33.13	-0.93
diamond2	1.84	11.67	3.77	20.10	3.77	22.86	15.53	27.43	15.53	22.98
diamond2	16.16	2.27	16.16	0.41	16.16	0.95	22.33	2.12	22.33	1.63
diamond2	23.90	3.65	23.90	2.85	23.90	0.52	23.90	3.78	23.90	1.31
diamond3	22.30	-2.84	27.75	-0.90	27.75	6.13	37.51	12.81	38.44	8.05
diamond3	28.65	-0.99	30.01	-1.75	30.01	-4.01	32.88	-1.69	33.09	-0.06
diamond3	18.45	3.30	18.69	1.90	18.35	6.67	25.55	4.74	23.99	0.68
diamond4	24.49	4.18	26.34	11.44	26.33	6.48	34.50	-10.77	31.95	0.20
diamond4	18.32	5.65	27.45	7.28	27.22	7.93	45.05	-0.21	45.51	6.27
diamond4	17.38	4.84	24.91	6.38	23.70	7.19	39.47	-5.50	43.15	11.49
divconq	17.13	-5.72	20.76	0.87	20.76	-2.52	23.60	-1.40	22.17	-6.51
divconq	15.97	5.56	18.52	-3.32	18.52	1.45	21.22	-0.07	19.43	-0.61
divconq	14.89	4.31	16.79	2.39	16.79	0.71	18.85	-0.67	17.50	-1.48
fft2	-6.99	-0.51	36.76	0.30	36.76	0.66	41.44	1.85	41.44	1.97
fft2	1.13	5.19	27.76	10.70	27.76	10.52	32.07	4.12	30.58	10.25
fft2	3.95	1.17	24.03	6.10	24.03	5.05	26.77	-5.12	26.04	4.59
gauss	-1.72	-6.47	-1.72	-6.46	-1.72	-6.49	12.34	16.88	6.46	2.19
gauss	25.59	4.22	25.59	2.58	25.59	4.09	25.59	1.53	25.59	2.91
gauss	34.72	1.53	34.72	5.15	34.72	5.16	34.72	-0.46	34.72	2.15
iterative2	-1.63	-5.88	-3.13	-6.05	-3.13	-6.05	5.28	1.92	0.21	-3.04
iterative2	-0.33	-1.04	0.18	-0.36	0.18	-0.37	0.18	-0.36	0.18	-0.36
iterative2	1.05	1.17	17.79	7.15	17.79	7.16	17.79	7.15	17.79	7.16
ms3	3.85	16.31	3.85	15.21	3.85	15.10	3.85	14.93	3.85	14.97
ms3	3.92	22.03	3.92	21.09	3.92	20.99	3.92	21.08	3.92	20.78
ms3	4.18	25.73	4.18	25.89	4.18	26.07	4.18	25.94	4.18	25.88
ms_gauss	6.47	2.16	27.73	11.09	26.89	1.13	38.87	5.21	46.45	5.25
ms_gauss	9.57	4.63	23.59	1.85	24.08	3.76	34.96	1.42	35.10	-0.21
ms_gauss	14.48	-0.27	16.91	-7.01	16.35	1.32	18.70	-8.92	17.56	-3.08
pde1	2.58	8.02	29.25	15.76	29.25	15.50	50.38	28.96	35.90	20.35
pde1	-6.17	3.88	14.80	5.02	16.24	4.15	38.71	8.22	23.17	11.12
pde1	-2.09	3.87	13.49	5.38	13.78	4.97	31.60	12.08	19.03	7.48
pde2	-5.33	9.97	11.20	4.72	11.96	4.22	38.30	8.79	21.48	11.63
pde2	5.65	13.32	18.28	9.80	17.69	10.78	32.68	14.71	24.81	12.83
pde2	12.62	8.53	20.77	3.50	20.59	4.36	29.26	8.19	26.00	8.39

Strategy	LPT		Quanti		Struc		Sim-an		Tabu	
	funct	exec	funct	exec	funct	exec	funct	exec	funct	exec
prolog	0.63	-0.92	1.15	-2.15	1.15	-2.15	2.08	-2.30	3.84	3.92
prolog	5.69	5.30	24.93	6.94	24.93	7.25	24.93	7.94	24.93	8.63
prolog	10.57	0.34	34.43	6.02	34.43	6.27	36.08	-8.66	36.03	-1.60
qcd2	0.00	4.96	0.00	4.78	0.00	5.20	0.00	5.64	0.00	3.34
qcd2	0.00	-1.48	0.00	2.92	0.00	2.45	0.00	0.76	0.00	2.97
qcd2	0.00	4.93	0.00	0.39	0.00	-1.30	0.00	0.83	0.00	-0.18
strassen2	9.17	9.53	15.37	22.87	15.37	18.61	24.67	12.55	25.28	9.62
strassen2	15.45	4.15	39.81	-7.09	34.88	2.75	39.81	-3.90	43.66	-2.40
strassen2	23.51	15.44	30.49	14.27	29.89	6.35	37.17	22.50	32.45	17.85
warshall	31.15	6.31	45.14	-1.46	45.14	-1.44	46.35	5.52	46.46	1.86
warshall	31.60	6.86	40.86	-1.50	40.86	-1.52	43.02	7.07	43.39	-2.47
warshall	28.57	5.59	34.49	-4.13	34.49	-4.13	37.42	-0.73	37.42	2.28

Gain par rapport au modulo avec comme coût Tore

Strategy	LPT		LGCF		Struc-LGCF		Sim-an		Tabu	
	funct	exec	funct	exec	funct	exec	funct	exec	funct	exec
bellford2	27.21	-1.02	26.60	3.72	27.72	12.80	26.60	3.95	26.60	3.63
bellford2	28.31	1.14	28.66	6.80	27.60	12.54	28.66	6.68	28.66	7.12
bellford2	18.71	1.19	35.73	2.41	16.77	12.11	35.73	2.51	35.73	2.54
diamond1	5.25	4.67	15.12	14.15	15.03	14.20	20.37	14.16	30.25	14.09
diamond1	26.23	-3.08	27.40	4.90	28.71	4.85	29.91	4.87	31.24	7.74
diamond1	31.05	-1.98	31.48	7.02	31.00	6.95	31.96	7.17	32.87	13.66
diamond2	25.05	7.16	18.83	25.90	25.96	26.19	36.02	13.21	44.04	28.08
diamond2	30.21	4.79	36.62	16.35	38.20	16.42	38.20	4.55	43.60	8.64
diamond2	6.24	3.01	-0.62	13.65	-9.65	9.46	25.89	15.45	32.13	7.75
diamond3	14.75	1.78	34.21	22.12	31.69	14.17	38.09	20.79	34.99	16.54
diamond3	24.68	-0.28	26.56	14.19	24.37	7.57	27.08	14.00	26.56	15.39
diamond3	17.20	-5.56	19.16	6.19	19.76	6.71	21.52	-3.36	19.16	6.63
diamond4	19.26	8.58	32.27	13.04	29.82	13.04	32.27	13.30	32.27	12.19
diamond4	3.85	8.26	31.23	16.74	29.92	18.74	35.39	15.09	31.26	15.83
diamond4	4.96	7.13	30.98	11.18	30.94	14.82	34.50	11.35	33.56	11.17
divconq	15.81	-1.73	21.50	3.55	21.50	1.36	21.50	4.12	21.50	3.51
divconq	16.05	1.46	19.09	7.17	19.09	5.03	19.47	7.33	19.09	7.49
divconq	14.74	0.94	17.65	6.65	17.65	5.85	17.65	7.31	17.65	7.08
fft2	-15.13	-4.13	33.69	4.28	33.58	4.07	33.69	4.27	33.69	4.30
fft2	2.25	0.18	32.47	10.04	31.44	9.38	32.47	10.07	32.68	10.06
fft2	1.79	0.18	24.78	10.27	24.48	10.80	24.78	10.28	24.78	10.45
gauss	-21.21	-3.10	-5.08	4.43	-5.08	2.71	18.36	-3.75	18.74	4.41
gauss	26.05	-8.23	26.05	-5.63	26.05	-6.48	30.92	1.81	30.30	-11.34
gauss	34.12	-2.87	34.12	-0.61	34.12	-1.93	34.12	-2.31	34.50	1.34
iterative2	-12.47	-2.08	-0.77	-2.45	-0.77	-2.45	-0.77	-2.45	10.78	-5.16
iterative2	-11.45	-2.18	-2.60	-6.34	-2.60	-6.34	-2.60	-6.34	13.90	2.50
iterative2	2.21	1.29	16.55	12.83	16.55	12.83	22.97	7.12	29.18	7.75
ms3	-28.83	16.39	2.19	-4.31	1.95	-4.32	2.19	-4.32	2.71	-2.01
ms3	-17.11	25.97	3.27	0.61	3.23	0.58	3.27	0.60	3.27	0.60
ms3	-1.62	22.71	4.17	-6.77	4.17	-6.75	4.17	-6.77	4.66	3.66
ms_gauss	-11.95	-9.15	35.17	1.48	21.46	10.46	35.17	1.37	35.17	1.49
ms_gauss	-1.80	-5.43	27.81	0.58	18.00	3.15	29.01	0.55	27.81	0.60
ms_gauss	13.45	6.13	17.83	3.88	17.12	9.09	17.83	3.83	17.94	3.87
pde1	0.44	8.48	39.89	25.97	23.59	25.15	39.89	24.74	39.89	25.97
pde1	-1.62	3.84	20.57	18.26	22.65	16.66	24.69	18.26	20.57	18.32
pde1	9.76	5.30	29.46	16.51	20.08	16.08	29.60	16.83	29.46	16.53
pde2	-0.03	10.69	25.32	15.47	18.11	15.14	25.32	15.81	25.32	15.72
pde2	10.43	10.80	26.44	16.65	21.31	14.42	26.44	16.49	26.44	16.64
pde2	11.26	13.39	23.23	15.41	19.43	16.20	23.23	15.30	23.23	15.49

Strategy	LPT		Quanti		Struc		Sim-an		Tabu	
	funct	exec	funct	exec	funct	exec	funct	exec	funct	exec
prolog	-13.05	-2.26	-3.07	-1.12	-3.07	-0.34	-3.07	-1.15	1.76	-1.53
prolog	6.15	3.26	24.04	10.59	24.04	12.41	26.85	3.80	24.04	10.65
prolog	11.64	-24.82	37.04	-13.68	37.04	-15.88	37.15	-13.03	37.04	-13.85
qcd2	-12.79	-0.89	-3.29	-1.07	-3.29	-1.14	5.85	-0.21	2.97	-1.24
qcd2	-7.94	-2.52	-2.05	-3.37	-2.05	-3.66	3.89	-5.74	3.89	-3.42
qcd2	-2.25	1.48	-0.58	-0.54	-0.58	-0.74	1.74	-0.75	0.52	-0.54
strassen2	9.74	1.87	18.20	21.87	26.25	21.50	18.20	21.75	36.19	16.84
strassen2	18.51	6.91	47.05	18.44	46.06	19.98	49.68	14.74	47.05	18.37
strassen2	17.42	3.51	30.74	12.57	29.27	8.62	30.74	12.75	30.74	12.67
warshall	24.72	7.43	39.85	0.49	31.96	6.05	39.85	1.89	39.85	0.86
warshall	25.66	9.00	38.69	6.82	27.18	4.08	38.69	6.82	38.69	6.82
warshall	26.26	3.80	34.44	2.46	26.63	1.09	34.44	2.46	34.44	2.47

Annexe C

Génie logiciel

C.1 Introduction

Le but de cette annexe est de mettre en évidence les développements logiciels effectués et les outils utilisés. En effet la plate-forme comprenant les outils de placement représentent plusieurs dizaines de milliers de lignes de codes et utilise des outils et langages tels que C, Perl, X-Window (XVIEW), YACC et LEX.

C.2 Interfaces

C.2.1 Langage de description de graphes

Une grammaire assez simple a été mise en place pour permettre de décrire les graphes de tâches. FLEX, l'analyseur lexical de GNU [1] a été utilisé afin de retrouvé les différents symboles clés et BISON de GNU a été utilisé afin d'agencer ces symboles en une grammaire.

Description de la grammaire

```
PROCESSEURS nombre_de_processeurs;
{taille_mémoire:proc_num{,proc_num};}
TACHES nombre_de_tâches;
{coût_de_calcul[,taille_mémoire]:tâche_num{,tâche_num}}
LIENS
{coût_de_communication:source->dest{,dest}}
```

Les symboles en majuscule doivent apparaître tels quels, tandis qu'aux termes en

[1] GNU: ensemble de produits du domaine public géré par la *Free Software Foundation*, 675 Mass Ave, Cambridge, MA 02139, USA

minuscules on doit substituer les valeurs correpondantes. Les accolades encadrent des parties qui peuvent être répétées autant de fois que nécessaires. les crochets encadrent des parties qui sont facultatives. Cette grammaire a été par la suite enrichie pour prendre en compte le type de machine et le diamètre du réseau de processeurs. De nombreux enrichissements de cette grammaire sont envisageables. Ils ne se sotn jusqu'à présent pas avéré nécessaires car la plupart des graphes de tâches utilisés ont été fournis automatiquement par des outils et non par des utilisateurs humains.

C.2.2 Interface X-Window

XVIEW 3.2, la bibliothèque de développement sous X-Window, de Sun, a été utilisée afin de fournir à l'utilisateur des outils de placement une interface conviviale(cf figure C.1). Les possibilités fournies comprennent : un éditeur de textes, un outil de sélection de fichier (*File Browser*, cf figure C.2) et divers menus de sélection de paramètres (type de machine cible, algorithmes de placement, temps maximum alloué au placement).

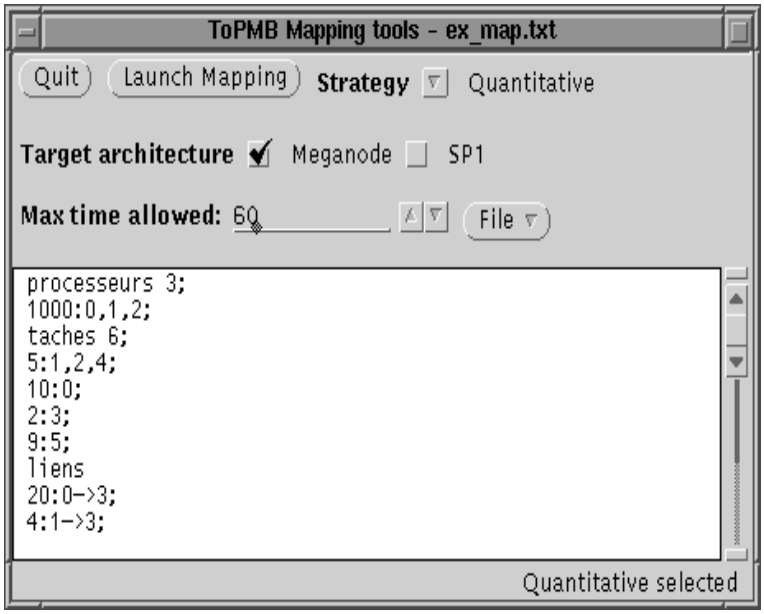
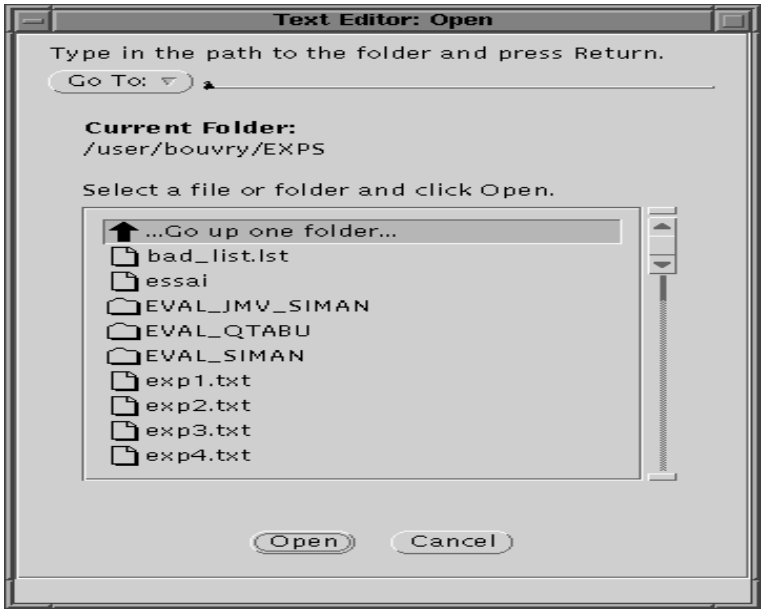


FIG. C.1 - : Interface X-Window



*

FIG. C.2 - : Choix d'un fichier à éditer

C.2.3 Table de routages

Le nombre de processeurs intermédiaires rentrant en jeu lors d'une communication donnée doit être connu pour certaines des fonctions de coût. Cette description peut être fournie aux algorithmes de placement sous forme d'un fichier ASCII[2] décrivant une matrice contenant ces valeurs. Un filtre traitant le fichier décrivant les tables de routage de VCR a été développé.

C.3 Fonctionnement interne

C.3.1 Bibliothèques

De nombreuses fonctions communes se retrouvent dans les différents algorithmes de placement : la gestion de matrices, l'analyse des fichiers de données, le principe des algorithmes de liste, la notion de relation de voisinage, le calcul des coûts de communication. Ces fonctions ont été regroupées dans différentes bibliothèques afin de faciliter leur maintenance.

[2] ASCII: *American Standard Code for Information Interchange*

C.3.2 Gestion de matrices creuses

Une gestion de matrices creuses (C.3) a été mise en place de manière à minimiser le temps d'accès aux éléments et la taille mémoire utilisée. En effet les graphes de tâches sont le plus souvent creux.

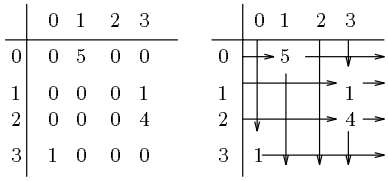


FIG. C.3 - : Gestion de matrices

Bibliographie

[ABPV89] S. Antonelli, F. Baiardi, S. Pelagatti, and M. Vanneschi. Communication cost and process mapping in massively parallel systems : a static approach. Rapport technique, Università degli studi di Pisa,dipartimento di informatica, 1989.

[Apa93] Apache. Présentation d'apache. Rapport technique 1, (LGI,LMC), Grenoble, Octobre 1993.

[Bae74] J. L. Baer. Optimal Scheduling on Two Processors of Different Speeds, pages 27–40. *Computer Architectures and Networks* North Holland, 1974.

[BBGT93] J. Błażewicz, P. Bouvry, F. Guinand, and D. Trystram. Scheduling complete intrees on two uniform processors with communication delays. LMC-IMAG, Rapport Interne, 1993.

[BBTW94] J. Błażewicz, P. Bouvry, D. Trystram, and R. Walkowiak. A efficient tabu search for the mapping problem. In *European Chapter on Combinatorial Optimization 95*. TU Poznan, Mai 1995.

[BCM+94] P. Bouvry, J. Chassin, M.Dobrucky, L.Hluchy, E. Luque, and T. Margalef. Mapping and load balancing on distributed memory systems. In *uP'94, Eight Symposium on Computer and Microprocessor Applications*, Octobre 1994. Budapest, Hongrie.

[BCR94] J. Briat, M. Christaller, and J-L Roch. Une maquette pour athapascan 0. In *RENPAR 6*. ENS Lyon, Juin 1994.

[BDSW91] J. Błażewicz, M. Drozdowski, B. Somiewicki, and R. Walkowiak. The two-dimensional cutting problem. Rapport technique CP-91-009, IIASA, Juin 1991.

[Ber89] F. Berman. Experience with an automatic solution to the mapping problem. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, 1989.

[Ber94] Stéphane Bernard. Notions d'analyse en composantes principales et application au choix de variables. *Tribunix - Dossier Benchmarks*, 10(55):88–100, Mai 1994.

[BESW93] J. Błażewicz, K. Ecker, G. Shmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, 1993.

[BGT94] P. Bouvry, J-M. Geib, and D. Trystram. *Analyse et Conception d'Algorithmes Parallèles*, chapitre Répartition de Charge. Livre publié par Hermes, avril 1994.

[BKPT94] P. Bouvry, Joao-Paulo Kitajima, B. Plateau, and D. Trystram. Andes: A performance evaluation tool, application to the mapping problem. Soumis à un numéro spécial de *Performance Evaluation*, 1994.

[BM88] S. W. Bollinger and S. F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. In *International Conference Parallel Processing*, 1988.

[Bok81] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30(3):207–214, Mars 1981.

[BP89] D. Benhamamouch and G. Plateau. Task allocation in distributed computing systems. Rapport technique, Univ. Paris-Nord, LIPN, Villetaneuse, 1989.

[Bra90] B. Braschi. *Principes de base des algorithmes d'ordonnancement de liste et affectation de priorités aux tâches*. Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble - France, Novembre 1990.

[BT89] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and distributed computation – numerical methods*. Prentice-Hall International, Englewood Cliffs, 1989.

[CC91] J-Y Colin and P. Chrétienne. CPM scheduling with small interprocessor communication delays. *Operations Research*, 39, 1991.

[CD72] E.G. Coffman and P.J. Denning. *Operating systems theory*. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[CG72] E.G. Coffman and R.L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[Chr89] P. Chrétienne. A polynomial algorithm to optimally schedule tasks on a virtual distributed system under tree-like precedence constraints. *European Journal of Operations Research*, (43):225–230, 1989.

[CK88] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, Février 1988.

[CT93] Michel Cosnard and Denis Trystram. *Algorithmes et architectures parallèles*. InterEditions, Paris, 1993.

[Dow93] Kevin Dowd. *High Performance Computing*. O'Reilly & Associates, Inc., 1993.

[dR94] Jean de Rumeur. *Ecole d'été sur les communications dans les réseaux de processeurs*. ERI. MASSON, 1994.

[ERL90] H. El-Rewini and T.G. Lewis. Scheduling parallel program tasks onto arbitrary target machine. *Journal on Parallel and Distributed Computing*, (9):138–153, 1990.

[FdK94] A. Fagot and J. Chassin de Kergommeaux. Optimized record-replay for rpc-based parallel programming. In *Working conference on programming environments for massively parallel distributed systems*, Ascona, Switzerland, Avril 1994. IFIP, WG10.3, Birkhaeuser, Basel.

[Fly66] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, décembre 1966.

[GGJ78] M.R. Garey, R.L. Graham, and D.S. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1):3–21, Janvier 1978.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman, New York, 1979.

[GL92] Fred Glover and Manuel Laguna. Tabu Search, a chapter in *Modern Heuristic Techniques for Combinatorial Problems*. W.H. Freeman, N-Y, 1992.

[Gra69] R.L. Graham. Bounds on multiprocessing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, Mars 1969.

[HB88] P. Haden and F. Berman. A comparative study of mapping algorithms for an automated parallel programming environment. Rapport technique, UC San Diego, 1988.

[HB93] Kai Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1993.

[Hey91] A. J. G. Hey. The GENESIS distributed memory benchmarks. *Parallel Computing*, 17(10–11):1275–1283, 1991.

[HK72] M. Hanan and J. M. Kurtzberg. A review of the placement and quadratic assignment problems. *SIAM Journal on Computing*, 14(2):324–342, 1972.

[HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on SSC*, 1968.

[Hu61] T.C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[IS90] Oscar H. Ibarra and Stephen M. Sohn. On mapping systolic algorithms onto the hypercube. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):48–63, Janvier 1990.

[Jai91] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley Professional Computing. John Wiley and Sons, New York, 1991.

[KB88] S.J. Kim and J.C. Browne. A general approach to mapping of parallel computations upon multiprocessor architectures. In *Int. Conf. on Parallel Processing*, volume 3, 1988.

[KCN90] Chung-Ta King, Wen-Hwa Chou, and Lionel M. Ni. Pipelined data-parallel algorithms: part II – design. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):486–499, Octobre 1990.

[Kit94] J. Kitajima. *Modèles Quantitatifs d'Algorithmes parallèles*. Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble - France, Novembre 1994.

[KN84] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, 1984.

[Kob78] Hisashi Kobayashi. *Modeling and analysis: an introduction to system performance evaluation methodology*. The Systems Programming Series. Addison-Wesley, 1978.

[KP92] João Paulo Kitajima and Brigitte Plateau. Building synthetic parallel programs: the project ALPES. In *Programming Environments for Parallel Computing*, éditeurs N. Topham, R. Ibbett, and T. Bemmerl, pages 161–170, Amsterdam, 1992. IFIP.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall Software Series. Prentice-Hall, Englewood Cliffs, deuxième édition, 1988.

[Lee91] C-L. Lee. Parallel machines scheduling with nonsimultaneous machine available time. *Discrete Applied Mathematics*, (30):53–61, 1991.

[Len93] J.K. Lenstra. Approximation algorithms for job shop scheduling. In *Workshop on Models and Algorithms for Planning and Scheduling Problems*, 1993. Atelier tenu à Come (Italie) en juin 93.

[LK90] Pei-Zong Lee and Zvi Mair Kedem. Mapping nested loop algorithms into multidimensional systolic arrays. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):64–76, Janvier 1990.

[Lo84] V.M. Lo. Heuristic algorithms for task assignment in distributed systems. In *Proc. 4th Int. Conf. Dist. Comp. Systems*, pages 30–39, 1984.

[Lo88] V.M. Lo. Algorithms for static task assignment and symetric contraction in distributed systems. In *ICPP88*, 1988.

[MGSK87] H. Muehlenbein, M. Gorges-Schleuter, and O. Kraemer. New solutions to the mapping problem of parallel systems: the evolution approach. *Parallel Computing*, (4):269–279, 1987.

[MRSV86] Debasis Mitra, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. Convergence and finite-time behaviour of simulated annealing. *Advanced Applied Probability*, 18:747–771, 1986.

[MS90] B. Monien and H. Sudborough. Embedding one interconnection network in another. *Computational Graph Theory, Springer-Verlag*, Computing Supplement 7, pages 257-282, 1990.

[MS91] Sridhar Madal and James B. Sinclair. Performance of synchronous parallel algorithms with regular structures. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):105–116, Janvier 1991.

[NT93] M. Norman and P. Thanish. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, Septembre 1993.

[Paz89] J. L. Pazat. *Outils pour la programmation d'un multiprocesseur à mémoires distribuées.* Thèse de doctorat, Université de Bordeaux I, 1989.

[Pea90] J. Pearl. *Heuristique : stratégies de recherche intelligente pour la résolution de problèmes par ordinateur.* Cépadues Edition, 1990.

[Per87] R.H. Perrot. *Parallel Programming.* Addison Wesley, 1987.

[Pop90] D. A. Poplawski. Synthetic models of distributed memory parallel programs. Rapport technique ORNL/TM – 11634, Oak Ridge National Laboratory – Martin Marietta, ORNL – Oak Ridge, Tennessee 37831 – USA, 1990.

[PR93] F. Pellégrini and J. Roman. Placement statique et bi-partition. In *Réunion CAPA de juin 1993.*

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity.* Prentice-Hall, 1982.

[RS87] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman - MIT Press, 1989.

[SC92] Chris J. Scheiman and Peter R. Cappello. A processor-time-minimal systolic array for transitive closure. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):257–269, Mai 1992.

[Sin87] J. B. Sinclair. Efficient computation of optimal assignments for distributed tasks. *Journal on Parallel and Distributed Computing*, (4):342–362, 1987.

[ST85] C.-C. Shen and W.-H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, C-34(3):197–203, Mars 1985.

[Sto77] H.S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(2):85–93, Janvier 1977.

[Tan88] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, 1988.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[TB91] E.G. Talbi and P. Bessière. Un algorithme génétique massivement parallèle pour le problème de partitionnement de graphes. Rapport technique, IMAG RR 845-I, 1991.

[TP94] Cécile Tron and Brigitte Plateau. Modelling of Communication Contention in Multiprocessors. In *Computer Performance Evaluation, Modelling Techniques and Tools*, éditeurs,Günther Haring and Gabriele Kotsis. Springer Verlag, Mai 1994.

[VRKL94] T. A. Varvarigou, V. P. Roychowdhury, T. Kailath, and E. Lawler. Scheduling in and out forests in the presence of communication delays. *a paraître dans IEEE Trans. on Parallel and Distributed Syst.*, 1994.

[WB93] Stephan Waser and Helmar Burkhart. OLGA - a modelling tool for algorithmic skeletons. In *Transputer Applications and Systems '93*, éditeurs,Reinhard Grebe et al. Volume 1, pages 395–409, Amsterdam, 1993. IOS Press.

[YG92] Tao Yang and Apostolos Gerasoulis. PYRROS: static scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 428–437. ACM, July 1992.

La demande croissante de puissance de calcul est telle que des ordinateurs de plus en plus performants sont fabriqués. Afin que ces machines puissent être facilement exploitées, les lacunes actuelles en terme d'environnements de programmation doivent être comblées. Le but à atteindre est de trouver un compromis entre recherche de performances et portabilité. Cette thèse s'intéresse plus particulièrement au placement statique de graphes de tâches sur architectures parallèles à mémoire distribuée. Ce travail s'inscrit dans le cadre du projet INRIA-IMAG APACHE et du projet européen SEPP-COPERNICUS (*Software Engineering for Parallel Processing*). Le graphe de tâches sans précédence est le modèle de représentation de programmes parallèles utilisé dans cette thèse. Un tour d'horizon des solutions apportées dans la littérature au problème de l'ordonnancement et du placement est fourni. La possibilité d'utilisation des algorithmes de placement sur des graphes de précédence, après une phase de regroupement, est soulignée.

Une solution originale est proposée, cette solution est interfacée avec un environnement de programmation complet. Trois types d'algorithmes (gloutons, itératifs et exacts) ont été conçus et implémentés. Parmi ceux-ci, on retrouve plus particulièrement un recuit simulé et une recherche tabu. Ces algorithmes optimisent différentes fonctions objectives (des plus simples et universelles aux plus complexes et ciblées). Les différents paramètres caractérisant le graphe de tâches peuvent être affinés suite à un relevé de traces.

Des outils de prise de traces permettent de valider les différentes fonctions de coût et les différents algorithmes d'optimisation. Un jeu de tests est défini et utilisé. Les tests sont effectué sur le Méganode (machine à 128 transputers), en utilisant comme routeur VCR de l'université de Southampton, les outils de génération de graphes synthétiques ANDES du projet ALPES (développé par l'équipe d'évaluation de performances du LGI-IMAG) et l'algorithme de regroupement DSC (*Dominant Sequence Clustering*) de PYRROS (développé par Tao Yang et Apostolos Gerasoulis).

Mots clés : Parallélisme, optimisation combinatoire, balancement de charge, placement, ordonnancement

The growing needs in computing performance imply more complex computer architectures. The lack of good programming environments for these machines must be filled. The goal to be reached is to find a compromise solution between portability and performance. The subject of this thesis is studying the problem of static allocation of task graphs onto distributed memory parallel computers. This work takes part of the project INRIA-IMAG APACHE and of the european one SEPP-COPERNICUS (*Software Engineering for Parallel Processing*). The undirected task graph is the chosen programming model. A survey of the existing solutions for scheduling and for mapping problems is given. The possibility of using directed task graphs after a clustering phase is underlined.

An original solution is designed and implemented ; this solution is implemented within a working programming environment. Three kinds of mapping algorithms are used: greedy, iterative and exact ones. Most developments have been done for tabu search and simulated annealing. These algorithms improve various objective functions (from most simple and portable to the most complex and architecturaly dependant). The weigths of the task graphs can be tuned using a post-mortem analysis of traces.

The use of tracing tools leads to a validation of the cost function and of the mapping algorithms. A benchmark protocol is defined and used. The tests are runned on the Méganode (a 128 transputer machine) using VCR from the university of Southampton as a router, synthetic task graphs generation with ANDES of the ALPES project (developped by the performance evaluation team of the LGI-IMAG) and the Dominant Sequence Clustering of PYRROS (developped by Tao Yang and Apostolos Gerasoulis).

Keywords : Parallelism, combinatorial optimization, load-balancing, mapping, scheduling