# Modèles quantitatifs d'algorithmes parallèles

Joao-Paulo Kitajima

THÈSE

présentée par

João Paulo Fumio Whitaker KITAJIMA

pour obtenir le grade de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

*(arrêté ministériel du 30 Mars 1992)*

(Spécialité : **Informatique**)

---

# Modèles Quantitatifs d'Algorithmes Parallèles

---

Date de soutenance : 20 octobre 1994

**Composition du jury**

*Président :*

JACQUES MOSSIÈRE

*Examinateurs :*

JEAN-MICHEL FOURNEAU (RAPPORTEUR)

JEAN-MARC GEIB (RAPPORTEUR)

JEAN-LOUIS PAZAT

BRIGITTE PLATEAU (DIRECTEUR DE THÈSE)

Thèse préparée au sein du
LABORATOIRE DE MODÉLISATION ET CALCUL – IMAG
et du
LABORATOIRE DE GÉNIE INFORMATIQUE – IMAG

# Remerciements

Je tiens à remercier **Jacques Mossière**, Professeur à l'Institut National Polytechnique de Grenoble, pour avoir présidé mon jury de soutenance.

Mes remerciements aussi à **Jean-Michel Fourneau**, Professeur à l'Université de Versailles, et à **Jean-Marc Geib**, Professeur à l'Université de Lille I, pour avoir accepté de juger ce travail. Leurs remarques ont été d'une grande importance pour mon travail.

Je tiens à remercier **Jean-Louis Pazat**, Maître de Conférences à l'Université de Rennes, pour avoir participé à mon jury de thèse et pour ses remarques.

Merci beaucoup à **Brigitte Plateau**, Professeur à l'Institut National Polytechnique de Grenoble, pour m'avoir donné le plaisir de développer un travail tout à fait innovateur. J'ai beaucoup appris pendant mon séjour à Grenoble et je le dois pour beaucoup à son esprit encourageant et très amical.

Merci à Gilles Villard, à Denis Trystram et à Jacques Chassin de Kergommeaux pour avoir vérifié la qualité de rédaction de la thèse.

Merci beaucoup au CNPq/Brésil pour son appui financier et pour sa confiance.

Enfin, je n'aurais rien pu faire sans l'appui de l'environnement très accueillant de l'IMAG et de la ville de Grenoble. Je remercie tout d'abord mes collègues de travail et amis, notamment Cécile Tron, Pascal Bouvry et Patrick Raynal. Merci aussi aux autres participants du Projet APACHE et de l'équipe de Calcul Formel du LMC-Centre Ville. Un merci spécial à Joëlle Prévost, Angèle Galindo, Liliane di-Giacomo et à Khadija Rassouli pour leur patience et leur sympathie. Je garde aussi un souvenir agréable du personnel du LGI-UGM, de la direction IMAG-INRIA et d'autres labos du Centre-Ville (principalement le LIFIA, le TIMA-Archi et le TIRF), sans oublier le LIP à Lyon et l'Ecole Doctorale de l'INPG (M. Adiba, Mme. Jourdan-Laforte et Mme. Vacher). Finalement, un merci à Jean Della-Dora et Yves Chiaramella, directeurs respectifs du LMC et du LGI en 1991, et au personnel de la Médiathèque pour leur accueil.

*Last but not the least*, un grand merci aux Etrangers (Brésiliens ou non) et aux Français que j'ai connu au Polygone Scientifique, à Xerox, SGS-Thomson, Grenoble Développement, au Shannon Pub et à la ville de Grenoble. Un souvenir sympa des copains de la Penduick FM et du programme "Chega de Saudade" de la Radio FM Kaleidoscope. Un merci spécial à Walcélio et à Keila, pour l'accueil en 91, à Rosa, à Javam, à Tia Momô et à Marcio pour les moments durs et pour le soutien, à Remis et à Alba, pour notre vieille amitié. Aux enfants Vico, Lucas, Levi, Felipe et Rafael. A PauloF, Ricardo Correa, Jaime, Fernando, Afonso et à Patrícia pour les moments "Brésil" au bureau et pour les repas ensemble. Aux nouveaux thésards qui arrivent. A Inés et à Monique, pour leurs sourires. A Claudia Linhares pour l'humour. A Anna, la Suédoise, pour la compagnie. Et surtout à Elliot, à Ana Tereza et à Luís Fernando, mes Parents et Frère : sans eux, l'amour et le bonheur n'auraient pas de sens. Un merci final aux "clans" des Whitakers et des Kitajimas pour le soutien, leurs coups de fil et leurs lettres.

*“This is an evolutionary process, and we hope that many more of you will write to us with your comments and suggestions about the current edition.” (Hillier & Lieberman)*

# TABLE DES MATIÈRES

# Table des figures

# Table des tableaux

# Chapitre 1

# Introduction

Le parallélisme est essentiel pour calculer plus vite. Dans la vie quotidienne, on se rend compte de que le travail en groupe est plus efficace que le travail individuel. Ainsi, toute une recherche existe autour de ce thème : développement d'algorithmes parallèles, d'ordinateurs parallèles et de systèmes d'exploitation parallèles. Pourtant, il y a un prix à payer pour cette nouvelle technologie. On espère que ce prix ne dépasse pas les gains apportés par le parallélisme. Le calcul de ce prix est complexe, car il inclut la satisfaction de l'utilisateur, la portabilité des applications séquentielles, le surplus d'énergie, par rapport à un ordinateur séquentiel, nécessaire pour maintenir une machine parallèle. Un calcul global de ce coût n'est pas envisageable, car presque impossible : sa décomposition selon différents aspects (techniques et humains; quantitatifs et qualitatifs) est une approche plus rationnelle.

Notre contexte est l'Informatique. Nos éléments de travail sont les algorithmes, les programmes, les ordinateurs et les systèmes d'exploitation. Il y a une multitude de ces éléments. Ainsi, toute **technique** qui permet la systématisation de l'évaluation du prix du parallélisme est bienvenue.

## 1.1 Objectifs

Le but de cette thèse est de présenter une **technique** générique de modélisation quantitative d'algorithmes parallèles et de l'utiliser dans un outil automatique d'évaluation des performances de systèmes (logiciel+matériel) parallèles. Cette approche consiste à exécuter des vrais programmes parallèles qui utilisent les ressources de la machine cible, mais qui ne résolvent pas un vrai problème de calcul (il s'agit de **programmes parallèles synthétiques**). Ces programmes sont tracés afin de mesurer leur performance. Cet outil est utilisé pour évaluer différentes stratégies de placement statique.

## 1.2 Motivation

La motivation de base est d'évaluer les gains et pertes du parallélisme. Les raisons spécifiques sont :

- déterminer les paramètres quantitatifs importants d'un algorithme parallèle;
- utiliser une technique d'évaluation des performances qui soit près de l'implémentation réelle, mais assez flexible pour permettre l'évaluation rapide de systèmes parallèles sous différentes charges de travail. L'outil doit permettre la **prédiction** des performances;
- développer et utiliser un outil automatisé pour l'évaluation des performances.

Le contexte de ce travail est le groupe *ALPES* (*AL*gorithmes *P*arallèles et *E*valuation de *S*ystèmes), lié au Projet APACHE (Algorithmique Parallèle et pArtage de CHargE), de l'IMAG (Institut d'Informatique et de Mathématiques Appliquées de Grenoble). *ALPES* est un groupe de recherche dont le sujet est l'évaluation des performances de programmes sur machines parallèles [Pla93]. Les autres axes de recherche dans *ALPES* sont :

- le traçage de l'exécution de programmes parallèles;
- la modélisation de machines parallèles;
- la visualisation des performances;
- la mise au point de programmes parallèles.

## 1.3 Algorithmes et programmes parallèles

Un des sujets de cette thèse est la modélisation d'**algorithmes parallèles**. Knuth [Knu73] définit un algorithme comme :

> "(...) étant un ensemble fini de règles qui donne une suite d'opérations afin de résoudre un type spécifique de problème."

Selon Knuth, un algorithme présente les caractéristiques suivantes :

1. un algorithme finit toujours;
2. les opérations d'un algorithme doivent être précises;
3. un algorithme a zéro, une ou plusieurs entrées;
4. un algorithme a une ou plusieurs sorties;

5. un algorithme est réalisable, c'est-à-dire, chacune de ses opérations peut être exécutée dans un temps fini par un être humain.

La définition d'algorithme donnée par Knuth n'est pas tout à fait adéquate pour les algorithmes parallèles, car il utilise le mot *suite*. Un algorithme est en effet un ensemble d'opérations nécessaires pour la résolution d'un problème. Cependant, la logique inhérente à un algorithme impose (ou pas) un ordre d'exécution à ses opérations. Cette logique peut même permettre l'exécution en parallèle de certaines opérations. Par exemple, supposons le problème suivant : quelle est la racine carrée de 3 plus 6? Pour résoudre ce problème, il y a un algorithme séquentiel : il faut calculer "3 plus 6", puis extraire la racine carrée de la différence. Supposons un autre problème : quelle est la racine carrée de 25 et la racine carrée de 36? On a alors 3 algorithmes possibles : deux séquentiels et un parallèle. Le premier algorithme séquentiel obtient $\sqrt{25}$ et puis $\sqrt{36}$. Le deuxième obtient $\sqrt{36}$ et puis $\sqrt{25}$. Enfin, l'algorithme parallèle est composé de cette opération "parallèle" : "faire simultanément $\sqrt{25}$ et $\sqrt{36}$".

Un **programme** est un algorithme exprimé dans un langage de programmation. Tous les programmes sont des algorithmes, mais pas tous les algorithmes sont des programmes. Un **programme parallèle** est un algorithme parallèle exprimé dans un langage de programmation parallèle. Il faut remarquer la différence entre un programme parallèle et son exécution. Un programme parallèle peut être exécuté en séquentiel (par exemple, par un ordinateur multiprogrammé). Par contre, un programme séquentiel n'est jamais exécuté en parallèle. Par la parallélisation automatique, un programme séquentiel est d'abord transformé dans un programme parallèle et puis exécuté (en parallèle ou en séquentiel).

Un algorithme est un ensemble d'opérations. Si on revient à la définition de Knuth, on remarque qu'il parle d'un "ensemble fini de règles" qui "donne une suite d'opérations". En effet, cet ensemble de règles constitue un **modèle de calcul** qui régit la construction d'algorithmes. Au niveau du programme, un modèle de calcul devient un **modèle de programmation**. Un modèle de calcul et son modèle de programmation respectif sont associés à un modèle de machine. Par exemple, dans les algorithmes séquentiels, le modèle de programmation le plus courant est associé à un ordinateur monoprocesseur avec une mémoire partagée par les opérations du programme. Par contre, dans les algorithmes parallèles, un problème se pose : il y a plusieurs modèles de calcul et, par conséquent, plusieurs modèles de programmation. Par exemple, dans [LER92], on trouve le modèle de programmation à mémoire partagée, le modèle de programmation à mémoire distribuée, le modèle de programmation à flot de données, le modèle de programmation par distribution de données, le modèle de programmation fonctionnel et le modèle de programmation par objets. En général, le choix du modèle de programmation dépend du type de l'algorithme, du système d'exploitation et de la machine parallèle disponibles. Dans cette thèse, les mots "algorithme" et "programme" sont utilisés de façon interchangeable.

## 1.4 Structure de la thèse

Le chapitre suivant fait un tour d'horizon sur les différents modèles d'algorithmes/programmes parallèles trouvés dans la littérature. Le chapitre 3 présente *ANDES*, notre modèle quantitatif. Ensuite (le chapitre 4), une technique permettant la construction de modèles *ANDES* est introduite (par le moyen de la bibliothèque *ANDES-C*). Cette technique peut être utilisée dans plusieurs méthodologies d'évaluation des performances d'algorithmes parallèles. Nous avons choisi de générer à partir de ce modèle des programmes synthétiques. Un outil (*ANDES-Synth*) a été développé en utilisant cette approche. L'outil est présenté dans le chapitre 5. Finalement, il est nécessaire de montrer une application réelle de l'outil développé, afin de valider notre travail. Le but du chapitre 6 est donc d'évaluer plusieurs stratégies de placement statique de tâches sur des multiprocesseurs à mémoire distribuée. Dans la conclusion, une analyse de notre travail est présentée avec les perspectives. Le Méganode (la machine cible sur laquelle l'outil s'exécute) est décrite en annexe.

# Chapitre 2

# Modèles d'Algorithmes Parallèles

Les systèmes informatiques parallèles et répartis deviennent de plus en plus présents. Autour de cet essor, toute une gamme de recherches, allant du domaine du matériel jusqu'au domaine du logiciel, est menée afin d'exploiter d'une façon la plus performante possible cette nouvelle technologie. Ces recherches entraînent souvent l'utilisation de **modèles**. Un modèle peut être considéré comme "toute structure logique ou mathématique formalisée, utilisée pour rendre compte d'un ensemble de phénomènes (dans notre cas, liés à l'informatique et au parallélisme) qui, bien que n'ayant pas de lien de causalité univoque, possèdent entre eux certaines relations" [Lar89]. Ces phénomènes peuvent être réels ou imaginaires et la façon d'obtenir un modèle dépend de certains aspects comme, par exemple, le but de ce modèle, le coût entraîné par la modélisation (soit en argent, soit en temps), et la facilité d'obtention de ce modèle. Avant tout, un modèle doit être considéré comme une abstraction qui nous aide à gérer et à comprendre la complexité inhérente des systèmes, en particulier, les systèmes informatiques parallèles.

Le but de ce chapitre est de présenter quelques techniques de modélisation d'algorithmes parallèles disponibles dans la littérature. Une analyse de toutes les techniques n'est pas désirable : nous présentons seulement les techniques les plus importantes et qui peuvent être utilisées dans un contexte d'évaluation quantitative des performances. Nous souhaitons ainsi modéliser une charge de travail (en anglais *workload*). Quelques aspects de chaque technique nous intéressent : *quels* phénomènes et relations sont pris en compte dans les modèles, *comment* les modèles sont-ils représentés, leurs *coûts d'obtention* et leurs *qualités* dans le contexte spécifique où chaque modèle est utilisé.

## 2.1 Modèles orientés vers l'ordonnancement

Prenons la définition du problème suivant : "ordonnancer en temps minimum un ensemble $T = \{T_1, \ldots, T_n\}$ de $n$ tâches soumises à des contraintes de précédence fixées (exprimées grâce à une relation d'ordre partiel, notée $<<$) sur $p$ processeurs identiques" [CT93]. Cette définition correspond au problème de **l'ordonnancement** de $n$

tâches d'un programme parallèle sur un ordinateur parallèle composé de $p$ processeurs. Un critère de qualité cherché par un algorithme d'ordonnancement $x$ peut être la minimisation du temps d'exécution de l'algorithme parallèle s'exécutant sur les $p$ processeurs (en anglais *makespan*). Un autre critère peut être l'équilibrage de l'utilisation des processeurs. Le concept de tâche varie d'étude en étude. Pour nous, un algorithme parallèle est composé de tâches (modules, activités ou opérations) qui éventuellement peuvent être exécutées en parallèle (cela dépend de la structure logique de l'algorithme). Une tâche a un ensemble d'entrées et un ensemble de sorties. Le travail de la tâche est modélisé par une fonction de traitement qui prend les entrées et produit les sorties. La granularité de ce travail (informellement la quantité d'instructions) peut être grosse (e.g., une procédure ou même un autre programme) ou petite (e.g., des opérations arithmétiques). La tâche est une unité de traitement indivisible. Enfin, la notion qu'il faut garder d'une tâche est qu'elle représente un calcul.

Nous portons un intérêt au problème de l'ordonnancement à cause des techniques de modélisation de l'algorithme parallèle associés. Les modèles utilisés sont des **graphes orientés sans circuit** (en anglais *Directed Acyclic Graphs* - DAG). Ces graphes sont valués afin de modéliser un coût de calcul (en temps ou en nombre d'opérations) de chaque tâche et un coût de communication entre les tâches (en temps ou en nombre d'octets). Un graphe orienté $G$ est un triplet $(S, A, \psi)$ où $S$ est un ensemble non vide de **sommets**, $A$ est un ensemble d'**arcs** et $\psi$ est une **fonction d'incidence** qui associe à chaque arc de $G$ une paire **ordonnée** de sommets (pas forcément distincts) de $G$ [BM81]. Un **circuit** dans un graphe orienté est un chemin simple de longueur $k$ ($k > 0$) dont le premier et le dernier sommets sont le même. Un **chemin** est une séquence non vide $s_0, a_1, s_1, a_2, s_2, \ldots, a_k, s_k$ (dont les termes sont alternativement des sommets et des arcs), telle que $a_k$ joint $s_{k-1}$ à $s_k$. $k$ est la longueur du chemin. Dans un **chemin simple** $C$, les arcs de $C$ sont distincts entre eux.

Dans les problèmes d'ordonnancement, le graphe orienté est souvent appelé le **graphe de précédence** de l'algorithme où les sommets modélisent les tâches et les arcs modélisent les précédences entre les tâches. Ainsi, dans un graphe de précédence, si $(a, b)$ est un arc qui **joint** le sommet $a$ au sommet $b$, alors la tâche modélisée par $b$ doit être exécutée après la tâche modélisée par le sommet $a$. Les coûts de calcul des tâches du graphe sont modélisés par une fonction $\theta(s)$ où $s \in S$. De façon analogue, les coûts de communication entre les tâches sont modélisés par une fonction $\gamma(a)$ où $a \in A$. $\gamma(a)$ modélise un coût de communication si les tâches ordonnées échangent des données. S'il n'y a que la précédence, ce coût dévient le prix payé pour envoyer un signal de fin d'exécution de tâche.

La Figure 2.1 présente un exemple de graphe de précédence. Le graphe modélise l'ordre d'exécution des opérations nécessaires à résoudre l'équation $ax^2 + bx + c = 0$, selon la formule :

$$x1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Dans cet exemple, les tâches ont une granularité très fine : ce sont une addition, une multiplication, une soustraction, une division, une négation ou l'obtention de la racine carrée. On suppose que toutes les données sont globalement disponibles pour chaque activité, que $a, b, c > 0$ et que $\sqrt{b^2 - 4ac} \geq 0$ (la mémoire est partagée).

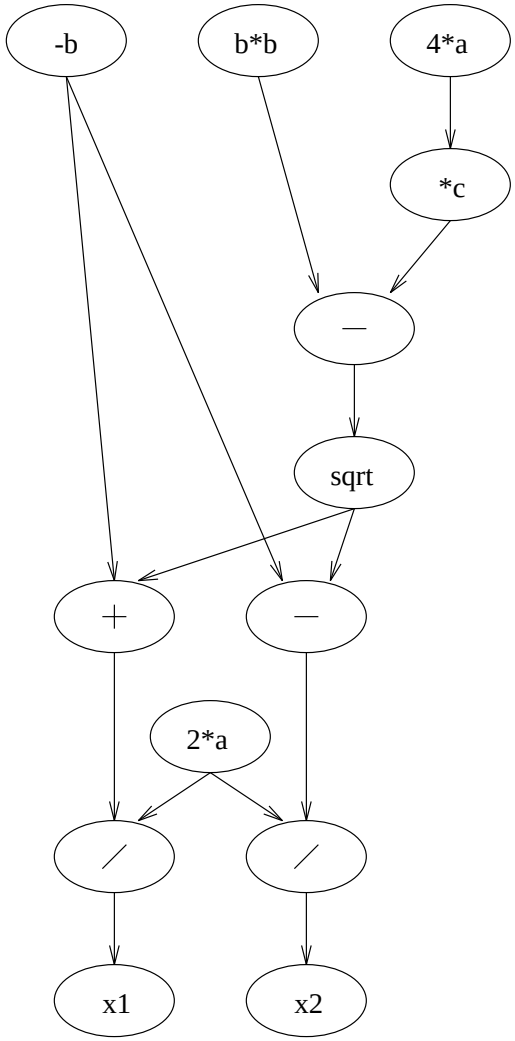



Figure 2.1 : Exemple de graphe de précédence.

Dans les premiers travaux sur l'ordonnancement de programmes parallèles, toutes les tâches ont le même temps d'exécution, étant donnée une machine cible, et la précédence ne représente pas un coût ($k$ = constante) :

$$\forall s \in S, \theta(s) = k$$

$$\forall a \in A, \gamma(a) = 0$$

Ce type de modèle (le même temps d'exécution pour toutes les tâches, sans coût de précédence) est nommé UET (*Unit Execution Time*). On le trouve dans [Hu61], [CG72], [RCJ72], [LS77] et beaucoup d'autres. A partir des modèles de type UET, des extensions ont été proposées : des activités avec des temps d'exécution différents [Ull75], [SH86], [Tow86] et des temps d'exécution aléatoires [Rob79], [BL90], [JBG90], modélisant des tâches avec un nombre variable d'opérations, non connu avant l'exécution de l'algorithme (e.g., une boucle dont la quantité d'itérations dépend d'une donnée d'entrée). Les modèles avec paramètres non déterministes font partie des études sur l'ordonnancement stochastique.

D'autres extensions du graphe de précédence concernent la communication entre les tâches. On trouve au départ des modèles avec un coût constant de communication (les travaux de Rayward-Smith, cité par [AM90]). Ces modèles sont connus sous le nom de UCT (*Unit Communication Time*). [H$^+$89] introduit des temps de communication qui sont fixes, mais différents entre eux. Son modèle est connu sous le nom de EDAG (*Enhanced Directed Acyclic Graph*). On trouve cette approche aussi dans [WG89] et dans le chapitre sur l'ordonnancement dans [LER92]. Gelenbe dans [Gel89] analyse le cas où les coûts de communication sont des moyennes. Il présente aussi une méthode probabiliste de construction de graphes en utilisant des descriptions hiérarchiques. La structuration d'un modèle en couches hiérarchiques permet une approche systématique de modélisation. GR (*Graphical Representation*) [Sar89] est autre exemple d'une technique hiérarchique basée sur les graphes de précédence.

En bref, les modèles utilisés dans les études du problème de l'ordonnancement représentent les **calculs**, la **précédence** entre les calculs, la **quantité de calcul** et la **quantité de communication** associée à une précédence. Ces informations sont modélisées par un graphe orienté sans circuit et valué. Il faut remarquer que, en général, ces quantités sont exprimées en unités de temps (temps d'exécution d'une tâche, temps de communication). Cela est possible si on connaît la machine parallèle cible. Si ce n'est pas le cas, les quantités sont exprimées, par exemple, en fonction du nombre d'opérations en arithmétique flottante ou fixe pour le coût de calcul des tâches et, par exemple, en fonction du nombre d'entiers, de chaînes de caractères et de réels pour le coût de la communication.

L'obtention de ces paramètres à partir d'un algorithme réel est un travail plutôt mécanique, sauf si certaines quantités ne sont pas connues avant l'exécution du programme : par exemple, des quantités qui dépendent des données. Dans ce cas, on peut travailler avec des moyennes ou des bornes, mais la détermination exacte de la distribution d'une quantité peut s'avérer très compliquée si l'algorithme est complexe. Il n'y a pas de technique bien établie pour l'estimation de ces paramètres et souvent on utilise l'intuition pour les obtenir.

Selon le paradigme de calcul, un graphe de précédence peut être assez complet (comme pour les programmes fonctionnels). Pour les autres modèles de calcul, certaines caractéristiques ne sont pas modélisés, comme les entrées et les sorties de l'algorithme.

Pourtant, les études sur l'ordonnancement exigent des modèles de programmes peu complexes au niveau de la quantité de caractéristiques modélisées, car le problème de l'ordonnancement est NP-difficile pour les graphes avec coûts déterministes et un nombre arbitraire de processeurs.

## 2.2 Modèles orientés vers le placement

Une autre interprétation donnée aux graphes orientés est celle où les arcs modélisent une communication entre tâches, mais pas une précédence : deux tâches peuvent s'exécuter en parallèle même si dans le modèle il y a un arc entre les deux sommets qui représentent ces tâches. Les graphes de communication ne sont pas forcément des graphes orientés sans circuit. Comme les graphes de précédence, les graphes de communication peuvent avoir des coûts constants ou aléatoires associés aux tâches et aux communications. En effet, les graphes de communication ont été beaucoup utilisés comme des modèles de programmes pour le problème du placement de tâches sur une machine parallèle [SB78], [Bok81], [FB89], [MT91], [A$^+$93]. Dans ces études de placement, les arcs du graphe de communication souvent perdent l'orientation, car pour les stratégies de placement, l'existence d'un canal de communication entre deux tâches est déjà une information suffisante. En fait, si $T$ est l'ensemble de tâches et $P$ l'ensemble de processeurs, un placement est défini comme "une application ($alloc$) qui à une tâche $t$ associe un processeur $p$ :

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

La recherche d'un placement se fait sur l'ensemble $P \times T$ de tous les placements possibles" [CT93]. Aucune hypothèse sur la précédence des tâches n'est faite. La Figure 2.2 représente un graphe orienté avec deux types de tâches : A et B. Tant qu'il y a du travail, la tâche A envoie, à chaque exemplaire de la tâche B, une partie de ce travail. Chaque exemplaire de la tâche B produit une partie du résultat qui est envoyée à la tâche A, qui compose le résultat final et, par exemple, l'affiche sur un écran. Ce schéma de programmation est connu sous le nom de "maître-esclave" (en anglais *process farm*).

Le graphe non-orienté correspondant au graphe de la Figure 2.2 est le graphe présenté dans la Figure 2.3.

Le graphe non-orienté est aussi connu comme **graphe de communication ou d'interaction**. Il contient moins d'informations que le graphe de précédence, mais il est adéquat pour le problème du placement (aussi NP-difficile pour un nombre arbitraire de tâches). Un graphe de communication modélise bien les programmes composés de processus séquentiels communicants (écrits, par exemple, en Occam2 [Lim88]).

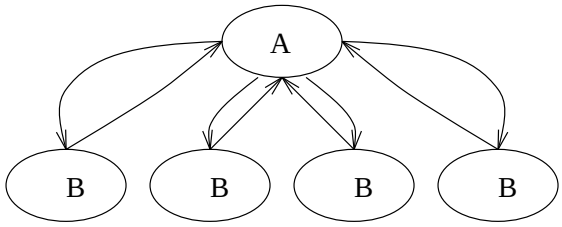



Figure 2.2 : Exemple de graphe de communication.

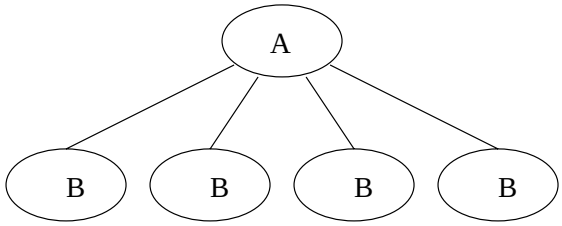



Figure 2.3 : Exemple de graphe de communication (non-orienté) utilisé dans les études de placement.

## 2.3 Extensions des modèles d'ordonnancement

### 2.3.1 Logiques d'entrée/sortie et probabilités

Des constructions d'algorithmes parallèles ne sont pas modélisées par les graphes de précédence et par les graphes de communication. Par exemple, on ne peut pas représenter toute structure qui dépend d'une décision (e.g., des boucles). Pour enrichir ces modèles de précédence, on a envisagé la définition, dans chaque tâche, de comportements d'envoi et d'acceptation de communications (ou **logiques de sortie et d'entrée**) (Figure 2.4).

Par exemple, certains travaux associent des opérateurs booléens à l'ensemble des communications d'une tâche. Par exemple, les graphes et/ou (*AND/OR*) analysés par Gillies & Liu [GL90] sont des graphes de précédence permettant :

- qu'une tâche démarre dès que l'un de ses prédécesseurs est fini (logique `ou`);
- qu'une tâche démarre si tous ses prédécesseurs ont fini (logique `and`).

La logique `ou` permet, par exemple, la modélisation d'acceptation non déterministe de communications. Dans le langage Occam2 [Lim88], cette acceptation est possible

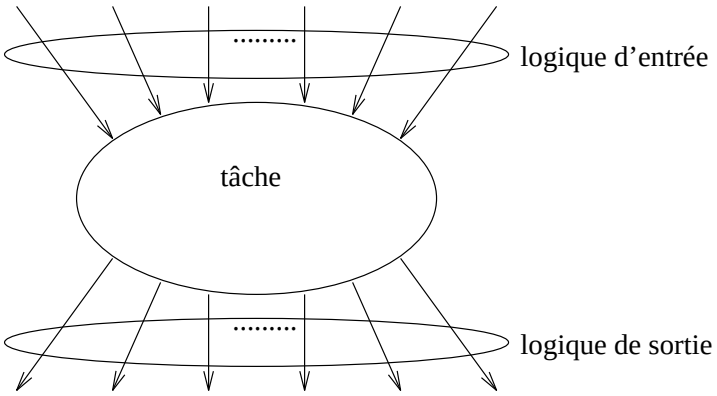



Figure 2.4 : Logiques d'entrée et de sortie.

grâce à l'instruction `ALT`. Gillies & Liu analysent le problème d'ordonnancer les programmes modélisés par ces graphes et/ou.

D'autres approches dans la littérature ajoutent des probabilités aux graphes orientés. Don Towsley [Tow86] étend le graphe de précédence de la façon suivante :

- à chaque arc $(a, b); a, b \in S$, il y a une probabilité associée $p_{a,b}$ qui détermine si la tâche modélisé par le sommet $b$ est exécutée ou pas après l'exécution de la tâche modélisé par $a$;
- à chaque tâche $t_i$ de l'algorithme, il y a une valeur associée $\mu_i$ qui donne la probabilité qu'aucune tâche ne sera exécutée après l'exécution de la tâche $t_i$;
- l'exécution de l'algorithme commence par la tâche $t_i$ avec probabilité $\lambda_i$.

Towsley permet que les graphes orientés aient des circuits, mais les méthodes d'ordonnancement présentées dans son article considèrent des graphes orientés sans circuit ou des graphes avec un nombre limité et connu de boucles. L'utilisation de probabilités révèle un certain souci de modélisation de l'influence des données sur le flot de contrôle de l'algorithme. La probabilité $p_{a,b}$ peut représenter une décision où chaque branche est choisie de façon indépendante ou elle peut représenter une structure plus commune comme le `PAR i=0 FOR n` (démarrage en parallèle de $n$ processus en Occam2) où la valeur de $n$ n'est pas connu avant l'exécution. La probabilité $\mu_i$ modélise un branchement possible à la fin du programme et la probabilité $p_{a,b}$ représente le choix d'un certain flot de contrôle selon les valeurs des entrées du programme.

Kapelnikov, Muntz & Ercegovac [KME89] mélangent les expressions booléennes avec les probabilités. Dans leur approche, les communications qui arrivent dans une tâche peuvent être du type `et` ou du type `ou`. En utilisant l'opérateur `union`, les communications d'entrée peuvent être groupées afin de modéliser des relations de

précédence plus complexes. La différence par rapport aux travaux de Gillies & Liu est l'existence de l'opérateur `union`. Pour les arcs de sortie d'un sommet, les opérateurs disponibles sont le `et`, le `ou-exclusif` et le `union`. Pour chaque arc de sortie, il y a aussi une probabilité associée. Pour la logique `et`, l'occurrence de chaque communication de sortie est indépendante des autres communications sortantes de la même tâche. La probabilité associée détermine si cette communication a lieu ou pas (cela correspond au $p_{a,b}$ de la technique de Towsley). La logique `et` permet la modélisation de constructions parallèles. Dans le `ou-exclusif`, seulement une communication est activée. La probabilité de cette activation est donnée par la valeur associée à l'arc correspondant. Cette logique modélise le choix (par exemple, les instructions `switch` et `if` de C). Finalement, le concept de segmentation est présenté. Un segment est un ensemble de tâches. Toutes les tâches d'un segment $SG$ qui ont de prédécesseurs externes à $SG$ sont démarrées en même temps et après que toutes les tâches des segments prédécesseurs aient fini. Ainsi, le graphe de segments est aussi un graphe de précédence de plus haut niveau hiérarchique. Kapelnikov, Muntz & Ercegovac utilisent ce modèle comme entrée d'un réseau de files d'attente. Tous les coûts sont des moyennes.

Les trois exemples présentés ci-dessus (Gillies & Liu, Towsley et Kapelnikov et al.) enrichissent le graphe de précédence présenté antérieurement. Ces techniques permettent la modélisation de structures contenant des décisions (par exemple, les boucles). La dernière méthode permet la description hiérarchique des modèles. Gelenbe dans [Gel89] présente aussi une méthode probabiliste de construction de graphes en utilisant des descriptions hiérarchiques. La structuration d'un modèle en couches hiérarchiques permet une approche systématique à la modélisation. GR (*Graphical Representation*) [Sar89] est autre exemple d'une technique hiérarchique basée sur les graphes orientés sans circuit.

Le choix du type de logique d'entrée ou de sortie lors de la modélisation est simple. Ce type est déterminé d'après le flot logique de contrôle de l'algorithme. Le grand problème est la détermination des valeurs des probabilités, soit dans les logiques de sortie, soit dans la construction hiérarchique probabiliste comme dans [Gel89]. En général, les travaux considèrent des graphes générés aléatoirement, où les probabilités ne modélisent pas une réalité particulière. Mais, étant donné un vrai programme, la dérivation de ces paramètres demande une analyse parfois complexe des interactions entre les entrées de l'algorithme et ses sorties. Comme pour l'obtention des coûts de calcul et de communication qui sont dépendants des entrées, l'obtention des probabilités est, en général, intuitive.

Nous remarquons encore que, dans les études sur la compilation et sur la parallélisation automatique de programmes séquentiels, on emploie souvent des graphes de précédence avec des logiques pour modéliser les flots de contrôle et de données d'un programme [ASU86], [BJP89], [P$^+$90], [Pol90], [LER92].

### 2.3.2 Hypergraphes

Une approche plus récente pour la modélisation d'algorithmes parallèles est celle présentée dans [LJ92]. Ce travail utilise des **hypergraphes**, une généralisation des graphes de précédence. Un hypergraphe orienté est un graphe dont les arcs représentent une relation $n$-aire. Dans les graphes classiques, cette relation est strictement binaire. La quantité de sommets engagés dans une relation n'a même pas besoin d'être fixe [Har88]. Les sommets du hypergraphe sont connus comme des **hypernœuds** et les arcs comme des **hyperarcs**. La Figure 2.5 montre un hypergraphe non-orienté.

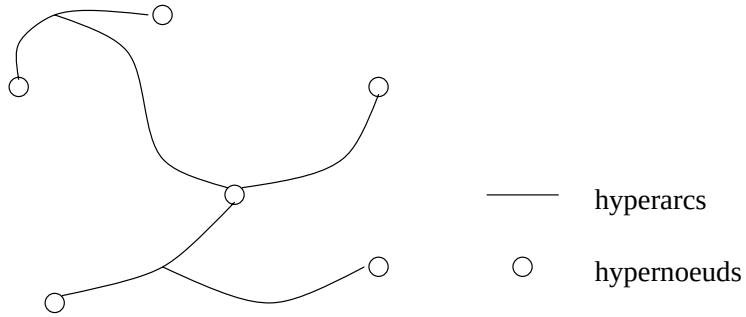



Figure 2.5 : Exemple de hypergraphe.

Li & Jamieson, dans [LJ92], utilisent les hypergraphes pour la modélisation des algorithmes parallèles pour le problème du placement utilisant la technique de comparaison de motifs (en anglais *pattern matching*). L'idée est de reconnaître la structure de dépendance de données de l'algorithme comme une instance d'une structure de dépendance pour laquelle on connaît déjà un algorithme de placement. L'outil de base est connu sous le nom de *Graph Matcher* et il est orienté vers les applications qui ont des structures régulières de dépendance de données. Etant donnée que les stratégies de placement implémentées dans *Graph Matcher* sont connues et qu'elles sont efficaces, le problème difficile reste la comparaison entre la structure du graphe de dépendance de l'algorithme original et les structures prévues dans l'outil. Cette comparaison est faite par un test d'isomorphisme. Dans ce contexte, les hypergraphes sont utilisés pour simplifier ce test. La méthode consiste à convertir un graphe de précédence classique en un hypergraphe, en permettant qu'un hypernœud modélise un ensemble de tâches et que ces hypernœuds soient connectés par des hyperarcs. Cette conversion doit être faite de telle façon que les propriétés du graphe de précédence original soient conservées dans l'hypergraphe associé. Le but est de réduire la taille du graphe de précédence à travers la représentation compacte des hypergraphes. La Figure 2.6 présente un graphe de précédence classique et son hypergraphe correspondant. Les valeurs dans les hypernœuds sont la cardinalité des ensembles des tâches associées et les étiquettes des hyperarcs décrivent la relation $n$-aire entre les hypernœuds (D = *distribute* correspond

à la relation 1-vers-$n$; P = *parallel* correspond à la relation $n$-vers-$n$ et M = *merge* correspond à la relation $n$-vers-1).

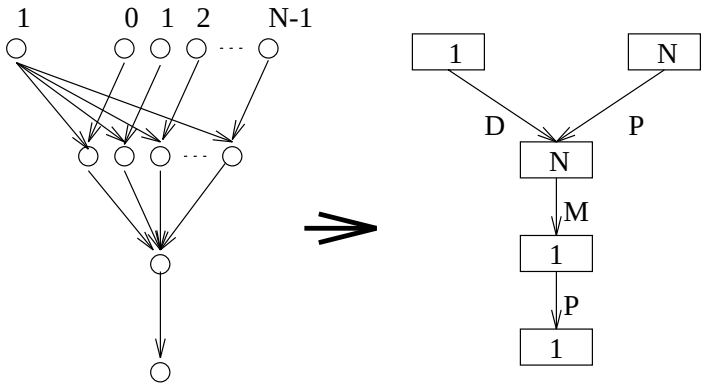



Figure 2.6 : Graphe de précédence et hypergraphe correspondant [LJ92].

Les aspects de l'algorithme parallèle modélisés avec les hypergraphes sont les mêmes que les aspects modélisés avec les graphes de précédence. Mais, les hypergraphes permettent une représentation plus compacte lorsque les graphes de précédence ont une régularité. Les communications globales (diffusion et rassemblement) sont explicitement modélises. Il faut remarquer que récemment on commence à parler sur les hypergraphes comme une méthodologie générale pour la modélisation de systèmes [dJ93] [Har88].

## 2.4 Modèles orientés vers la conception

### 2.4.1 GMB – *Graph Model Behavior*

GMB (*Graph Model Behavior*) est un formalisme développé dans le projet SARA (*System ARchitects Apprentice*) [E$^+$86]. L'objectif du projet SARA est très général : le développement d'une méthodologie interactive de programmation et d'un environnement d'aide à la conception, à la modélisation, à la simulation et à l'analyse de systèmes. Ces systèmes sont modélisés en utilisant deux langages :

- **SL** (*Structure Language*) : utilisé dans la construction de la structure du système (son architecture). SL permet des descriptions hiérarchiques. Ces primitives sont le module, les prises (en anglais *sockets*) et les interconnexions;

- **GMB** : utilisé dans la modélisation du comportement du système. Trois différents domaines sont analysés : le contrôle, les données et l'interprétation.

Carpenter & Tyrrell [CT89] utilisent GMB en particulier pour la modélisation d'algorithmes. Leur objectif est d'employer GMB pour la conception de logiciels tolérants aux pannes pour les systèmes répartis. Les aspects contrôle et données de l'algorithme sont modélisés en utilisant des graphes orientés. Le graphe de contrôle est un graphe de précédence étendu avec des logiques d'entrée et de sortie. Le flot de contrôle est représenté par des jetons mobiles sur le graphe.

Le graphe de données est aussi orienté. Il y a deux types de sommets : ceux qui modélisent les processeurs de données et ceux qui modélisent les données. Chaque processeur de données est associé à une activité qui utilise des données. Dans GMB, il y a une correspondance univoque entre les processeurs de données et les activités du graphe de contrôle. Les arcs modélisent le sens d'accès aux données : un arc du processeur de données vers les données représente une écriture. Le sens inverse modélise une lecture.

Le domaine de l'interprétation est caractérisé par un langage spécifique qui permet de décrire l'association entre le domaine de contrôle et le domaine des données. Il n'est pas nécessaire de décrire toutes les opérations modélisées par un sommet du graphe de contrôle, mais, au moins, il est nécessaire de décrire les conditions analysées, par exemple, pour déterminer quelle sortie doit être activée dans une décision. Un exemple d'un modèle GMB est présenté dans la Figure 2.7. Dans cette Figure, le nœud Q s'exécute après le nœud A et avant les nœuds B , C et D. Q a une logique de sortie du type `OU` (représenté par le symbole $+$), c'est-à-dire, que B ou C ou D s'exécute. Dans le domaine des données, Q **lit** la donnée `data`. Le domaine de l'interprétation montre que, dans le nœud Q, `value` reçoit la valeur de `data`. Si `value` est égal à 0, la sortie `y` est activée. Si `value` est plus grand que 0, c'est la sortie `x` qui est activée. Sinon, c'est la sortie `z` qui a lieu.

Les trois domaines sont modélisés de façon distincte. Une représentation unique est possible, mais peu désirable si les modèles sont complexes. La proposition originale de GMB ne propose pas d'utiliser des annotations de coûts. Pourtant, les informations de charge peuvent être facilement fournies soit en modifiant le langage de description dans le domaine de l'interprétation, soit en utilisant des fonctions similaires à $\gamma$ et à $\theta$ existantes dans les graphes des paragraphes antérieurs. Une remarque importante est que les coûts de communication correspondent à la taille des structures de données utilisées par les processeurs de données. Etant donné que les modèles GMB développés par Carpenter & Tyrrell sont des transcriptions de vrais programmes et que GMB supporte une modélisation très précise de la réalité (grâce au domaine de l'interprétation), l'obtention des modèles est simple. La nouveauté de GMB par rapport aux graphes de précédence avec logiques d'entrée et de sortie est la description plus complète des structures de données et leur utilisation par les tâches.

### 2.4.2 ELGDF – *Extended Large Grain Data Flow*

ELGDF est un langage graphique de description développé par El-Rewini & Lewis [ERL88]. Ce langage est utilisé par l'environnement PPSE (*Parallel Programming*

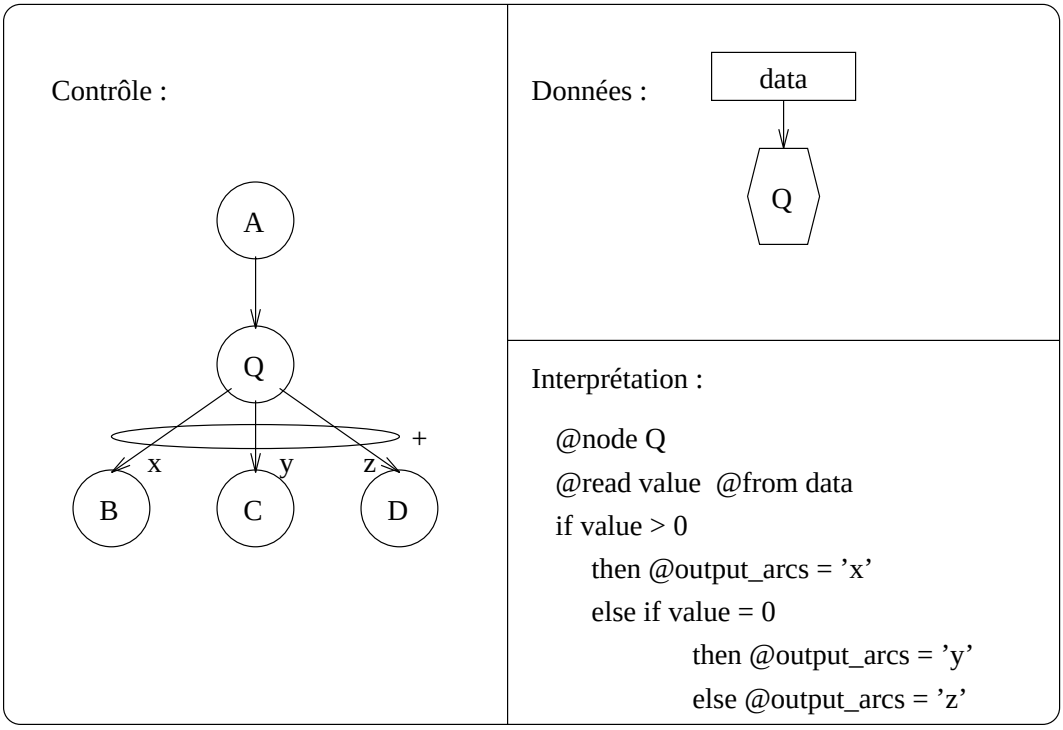



Figure 2.7 : Exemple de modèle GMB.

*Support Environment*) pour la description des applications parallèles. ELGDF a été conçu pour être utilisé dans un environnement d'aide à la programmation parallèle. Il s'agit d'une extension des graphes LGDF (*Large Grain Data Flow*) [Bab84]. Comme dans GMB, les modèles en ELGDF sont des graphes orientés avec des sommets représentant soit des activités soit des données. Les arcs peuvent modéliser une dépendance de contrôle (précédence) ou une dépendance de données. Il y a aussi des sommets spéciaux qui modélisent les décisions existantes dans un algorithme (la Figure 2.8 montre le sommet de décision `split` basé sur une condition f(X) où X est une donnée).

ELGDF se caractérise aussi par l'existence de "meta-structures" comme les répétiteurs, les boucles, les pipelines. Ces structures sont construites à l'aide d'opérateurs. ELGDF permet des descriptions hiérarchiques et la modélisation de stratégies d'accès aux données partagées. A partir d'un graphe ELGDF, un graphe de précédence peut être dérivé. Les coûts de calcul et de communication sont obtenus à partir d'estimations. Le graphe de précédence est converti en un programme écrit en C-Linda, Fortran ou Strand. L'exécution du programme est tracé en utilisant un outil nommé EPA (*Execution Performance Analysis*). L'implémentation de ELGDF s'appelle Parallax [KL90].

Le coût d'obtention d'un modèle ELGDF dépend de la complexité du programme. PPSE est un environnement de support à la programmation parallèle. L'idée est que

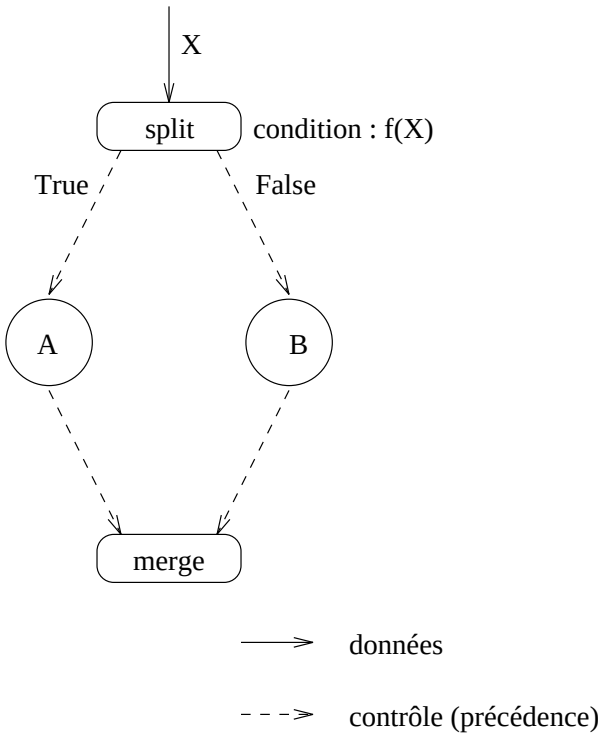



Figure 2.8 : Exemple de décision dans ELGDF.

les coûts de programmation en utilisant PPSE soient plus faibles que les coûts sans l'utiliser. Les structures particulières à ELGDF sont les "meta-structures" qui aident la construction du modèle, les arcs qui contiennent des informations sur la résolution des conflits d'accès aux données et les deux types d'arcs de flot de données et de flot de contrôle.

### 2.4.3 ParaDiGM

ParaDiGM [DSN88] (*Parallel Distributed computation Graph Model*) est une méthodologie développée pour l'évaluation qualitative de différentes stratégies d'implémentation d'algorithmes parallèles, selon un modèle de programmation basé sur l'échange de messages. ParaDiGM est composé de deux techniques complémentaires de modélisation : le PAM (*Process Architecture Model*) et le DCPG (*Distribution Computation Precedence Graph*) :

1. un graphe PAM décrit les activités (les processus) et les communications existantes entre ces activités. Il est similaire à un graphe de communication. Les activités sont modélisées par des cercles et la communication entre activités, par des carrés. L'utilisation d'ellipses dans le graphe PAM permet la représentation

d'une relation de communication avec un nombre d'activités identiques qui ne sont pas toutes décrites dans le graphe. Dans la Figure 2.9, il y a deux types d'activités : `main` et `secondary`. Il y a une seule instance de `main` et plusieurs de `secondary`. Il y a deux types de communications :

(a) une communication globale entre `main` et les `secondary` (`el-1` dans la Figure 2.9);

(b) plusieurs instances de la communication point-à-point, chacune reliant `main` à chaque exemplaire de `secondary` (`el-2` dans la Figure 2.9).

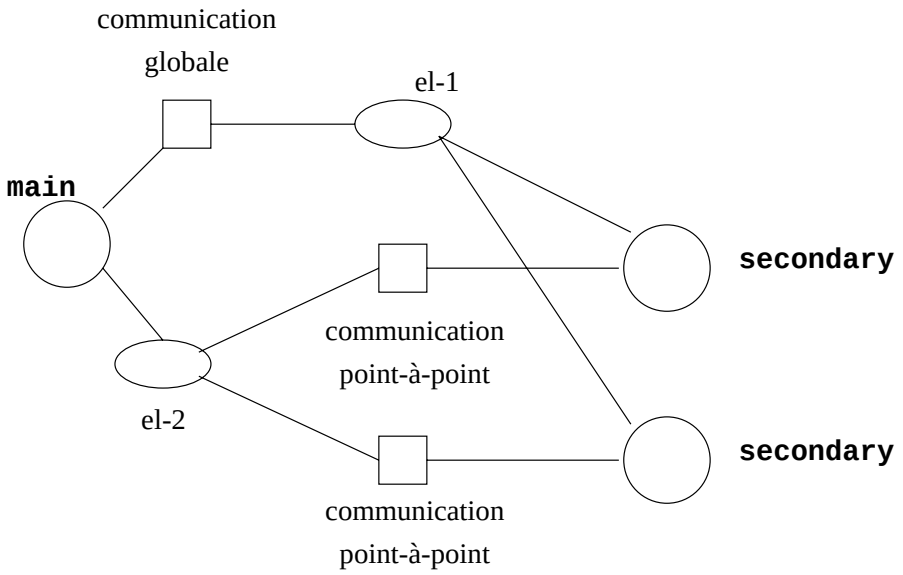



Figure 2.9 : Exemple d'un graphe PAM [DSN88].

2. un graphe DCPG (Figure 2.10) modélise le flot de contrôle des activités. Il ressemble à un graphe de précédence. Les constructions principales sont :

   - les grands cercles qui modélisent une tâche de calcul;
   - les petits cercles blancs qui modélisent des alternatives (nœuds `or`);
   - les cercles noirs qui représentent la création d'un processus (nœuds `spawn`);
   - les ellipses qui modélisent la création de plusieurs processus identiques (nœuds `multiple spawn`);
   - les carrés représentent un "dépôt" de données (soit un "dépôt" local, e.g. une variable, soit un message);

- le losange modélise la jonction de plusieurs arcs de contrôle;
- les triangles modélisent le début et la fin d'un processus.

Il y a une correspondance entre le PAM et le DCPG. Par exemple, la Figure 2.10 montre le DCPG correspondant au processus `main`. La tâche 2 démarre les instances des tâches `secondary` (voir Figure 2.9). Le cercle 3 modélise la création du processus `main`. La tâche 4 fait une diffusion (en anglais *broadcast*) de la donnée `m1` et écrit dans `r1` (cette diffusion correspond au carré "communication globale" du PAM). La tâche 6 fait un envoi sélectif (soit `m2`, soit `m3`) et lit aussi la donnée `r1`. La tâche 7 fait une réception sélective (un récepteur et plusieurs possibles émetteurs) et écrit dans la donnée `r2`. Les communications associées aux tâches 6 et 7 correspondent aux carrés "communication point-à-point" du PAM. Le nœud 8 modélise une décision du type `if-then-else` (re-exécute les tâches 6 et 7 ou exécute le nœud final). Le nœud 8 lit aussi la donnée `r2`.

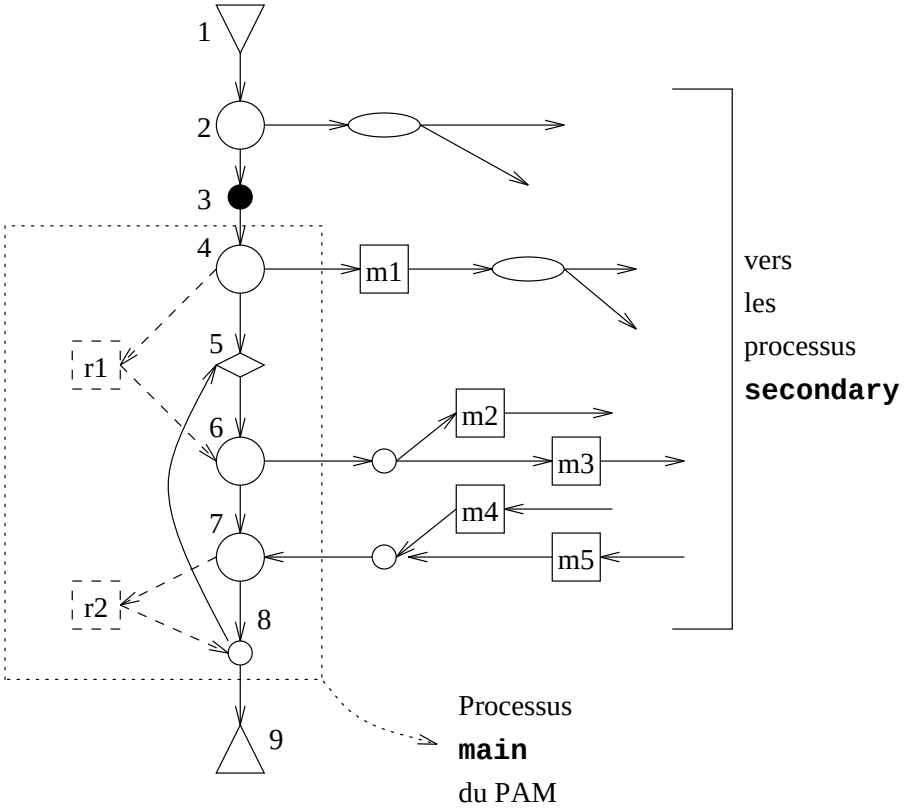



Figure 2.10 : Exemple d'un graphe DCPG [DSN88].

Les modèles de ParaDiGM sont utilisés pour une analyse plutôt qualitative des algorithmes parallèles. Des fonctions numériques représentant les coûts ne sont pas

prévus, quoique leur implémentation dans le modèle soit facile. Un élément nouveau dans ParaDiGM par rapport aux approches antérieures est la possibilité de modélisation des opérations globales comme la diffusion des données. On peut modéliser aussi la génération (récursive ou non) des processus (*spawn*). Les échanges bloquants et non bloquants peuvent être aussi représentés. Dans les graphes PAM, il est possible aussi de modéliser des classes de processus et des instances de ces classes. A cause du grande nombre de structures, un modèle ParaDiGM peut être très complexe et coûteux.

## 2.5 Modèles orientés vers la spécification

### 2.5.1 Réseaux de Petri stochastiques

Les réseaux de Petri [FP89] sont des outils très connus pour la modélisation de systèmes. Au départ, ils sont utilisés pour la validation formelle de systèmes. Un réseau de Petri est défini comme un quadruplet $R = \{P, T; Pre, Post\}$ où :

- $P$ est un ensemble non vide et fini de **places**;
- $T$ est un ensemble non vide et fini de **transitions** avec $P \cap T = \emptyset$;
- $Pre$ est une fonction $P \times T \rightarrow N$ nommée d'**incidence avant**. Elle détermine le poids de l'arc reliant une place à une transition ($N$ est l'ensemble de naturels);
- $Post$ est une fonction $T \times P \rightarrow N$ nommée d'**incidence arrière**. Elle détermine le poids de l'arc reliant une transition à une place.

Un réseau de Petri marqué est déterminé par le couple $RM = \{R; M\}$ où $M$ est le **marquage** du réseau. $M$ est une fonction $P \rightarrow N$. En bref, le fonctionnement d'un réseau de Petri est basé sur les conditions de franchissement des transitions du réseau. Une transition est **franchissable** si et seulement si $\forall p_i \in P, M(p_i) \geq Pre(p_i, t_k)$ (si la place $p_i$ est reliée à la transition $t_k$). Si une transition est franchissable, alors cette transition peut avoir lieu (pas forcément elle sera franchie). Si plusieurs transitions du réseau sont franchissables, alors une seule transition se produit à la fois selon un choix non déterministe. Le franchissement d'une transition $t_k$ conduit à un nouveau marquage $M'$ donné par :

$$\forall p_i \in P, M'(p_i) = M(p_i) - Pre(p_i, t_k) + Post(t_k, p_i)$$

Avec les réseaux de Petri stochastiques, l'évaluation des performances en utilisant ce formalisme est devenue possible. En effet, les réseaux de Petri stochastiques permettent une analyse quantitative des réseaux de Petri classiques, en ajoutant des informations sur :

- les temps de franchissement des transitions. Ces temps sont définis par des distributions de probabilités;

- les conflits structurels. Par exemple, lorsque deux transitions sont franchissables à partir d'une même place, il faut définir des probabilités de transition afin de quantifier ce choix.

Un exemple de modélisation d'algorithmes par des réseaux de Petri est présenté dans [JM92]. Dans cet article, les algorithmes sont écrits en Occam2 [Lim88] puis traduits en réseaux de Petri. Une place modélise un état spécifique de l'algorithme. Si cette place est marquée, alors l'état correspondant a été atteint. Le franchissement d'une transition modélise l'exécution d'une ou plusieurs instructions de l'algorithme. Par exemple, la Figure 2.11 présente une partie d'un programme Occam2 et le réseau de Petri correspondant. Il y a deux processus exécutant simultanément : Q1 et Q2. Q2 envoie la valeur 10 par le canal c et Q1 reçoit cette valeur par le canal c et le met dans x. Dans le réseau, p1 et p2 représentent l'état de chaque processus avant la communication. p3 et p4 modélisent les états de chaque processus après la communication. t1 représente la communication.

```
PAR
  SEQ    -- processus Q1
    c?x
    SKIP
  SEQ    -- processus Q2
    c!10
    SKIP
```

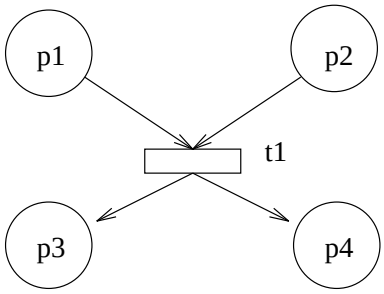



Figure 2.11 : Exemple de modélisation de programmes en Occam2.

Un autre exemple de modélisation par réseaux de Petri est présenté dans [B+92]. Il faut remarquer qu'un modèle de programme parallèle décrit comme un réseau de Petri représente au même temps le programme et la machine parallèle.

Les réseaux de Petri sont une technique de haut niveau pour la modélisation de systèmes. Ainsi, il peut avoir plusieurs modèles pour le même programme. Ces réseaux sont adaptés à la modélisation des aspects dynamiques de systèmes, tels quel la synchronisation, la contention, etc…. La qualité des modèles est variable, très dépendante des contextes dans lesquels ils sont utilisés. Les réseaux de Petri sont très adéquats lorsqu'on veut valider formellement le modèle (e.g., détecter l'occurrence d'une situation d'interblocage).

### 2.5.2 Files d'attente et modèles de programmes

Les modèles basés sur les réseaux de Petri représentent le programme parallèle et la machine parallèle, sans une claire séparation entre ces deux éléments. Les files d'attente sont une autre possibilité de modélisation composée de clients et de serveurs connectés entr eux. Les clients utilisent les serveurs. Une file d'attente est associée à chaque serveur. Les clients restent dans cette file si le serveur est occupé (c'est-à-dire, s'il sert un client). A la fin du service, le client servi peut sortir du système ou peut se diriger à un autre serveur. La Figure 2.12 présente un exemple simple d'un réseau de files d'attente. Dans la Figure, les clients arrivent par $A$. Ils passent par la station de service 1 et par la station de service 2. Avec probabilité $p_0$, ils sortent du système. Avec probabilité $1 - p_0$, ils passent par la station 3 et retournent à la station 1.

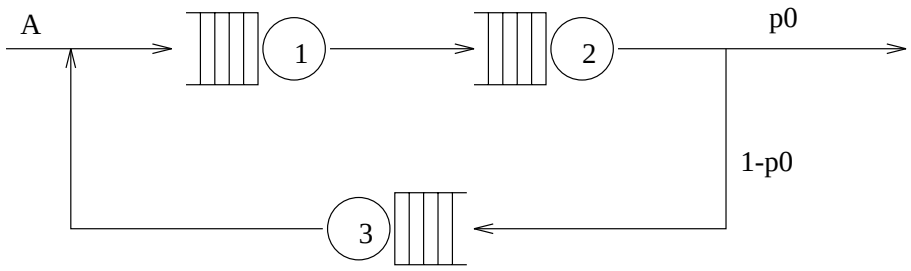



Figure 2.12 : Réseau de files d'attente.

Dans la modélisation de programmes parallèles et de machines parallèles, les clients représentent les programmes et les serveurs représentent les différents composants de l'architecture de la machine (les processeurs, la mémoire, le réseau d'interconnexion). En fait, les programmes modélisent la charge de travail et les serveurs modélisent les ressources. Les clients sont caractérisés principalement par :

- leur nombre;
- la variable aléatoire qui décrit leurs arrivées;
- leurs demandes de service.

D'autres caractéristiques peuvent être considérées comme la priorité. Ces caractéristiques sont en général dérivées du programme parallèle ou de son modèle (par exemple, d'un graphe de précédence). Les clients dans un réseau de files d'attente sont anonymes. Leur précédence est définie par l'ordre d'arrivé et de service, mais cette précédence ne peut pas être forcée d'avance. Des hiérarchies de files d'attente peuvent être construites afin d'aider l'analyse du système.

## 2.6 Modèles orientés vers la simulation

### 2.6.1 Modèles *MIMD*

*MIMD* (*Multiple Instruction stream, Multiple Data stream*) [CS92] est un système de simulation développé sur *DEMOS*, un logiciel de modélisation pour la simulation discrète écrit en Simula. *MIMD* s'exécute sur des postes de travail Sun. Cet outil supporte la simulation de l'exécution de programmes parallèles basés sur le modèle de passage de messages sur des multiprocesseurs à mémoire distribuée. Le programme parallèle (supposé être correct) est modélisé comme un graphe orienté dont les sommets représentent les processus et les arcs modélisent des canaux unidirectionnels. Un processus peut exécuter seulement 4 types de instructions : $compute(n)$ (calculer en utilisant $n$ cycles du processeur), $sleep(n)$ (dormir pendant $n$ secondes), $send(C_i, n)$ (envoyer un message de $n$ octets par le canal $C_i$) et $receive(C_i)$ (recevoir un message sur le canal $C_i$). Le code ci-dessous présente la description d'un processus qui reçoit un message sur le canal $C1$ (ligne 27), réalise un calcul synthétique (ligne 28) et envoie un message de 512 octets (ligne 29) par le canal $C2$ :

```
(...)
23  Model1_Subprocess Class recvFirst(C1);
24  ref(Channel) C1,C2;
25  Begin
26  While TRUE do
27  Begin Receive(C1);
28        Compute(100);
29        Send(C2,512);
30  End;
31  End;
(...)
```

La machine parallèle est représentée par un graphe non orienté dont les sommets modélisent les processeurs et les arêtes modélisent les liens de communication. Tous les processeurs sont du même type (architecture homogène). Un extrait d'une description de matériel est présenté ci-dessous. La ligne 36 montre que le processeur `ECS` a une tranche de multiprogrammation de 5000 unités de temps par processus, un débit de 0 pour un canal logiciel (intra-processeur), un débit de 3/40 pour un canal physique et une stratégie de routage 0 (i.e., le temps écoulé entre l'arrivé d'un message et son départ sur le même processeur est considéré nul). La topologie physique est nommée `Meiko` et il s'agit d'une grille 2x2 (ligne 37) :

```
(...)
36  ECS :- New Hardware_Description("ECS",5000,0,3/40,0);
37  Wiring_structure :- NewPatternMesh("Meiko",2,2);
```

```
(...)
```

Le placement des processus sur les processeurs est fait par le moyen de procédures spécifiques. La simulation produit des performances comme, par exemple, le taux d'utilisation des processeurs et des liens et l'activité des processus et des canaux. Des portraits instantanés du système simulé peuvent être aussi obtenus sans perturber l'exécution simulé.

Dans *MIMD*, les modèles quantitatifs de programmes parallèles sont similaires aux graphes utilisés pour les algorithmes d'ordonnancement. Cependant, dans *MIMD*, les modèles sont associés à un paradigme de programmation très spécifique (passage de messages) avec la possibilité de modéliser la suspension de l'exécution d'un programme parallèle (par le moyen de l'instruction *sleep*).

### 2.6.2 *HAMLET*

*HAMLET* est un environnement automatique développé pour aider le projet d'applications industrielles parallèles. Un accent est posé sur les premières étapes du projet, car "plus tard un erreur est détecté, plus chère est sa réparation" [PPJ94]. *HAMLET* est composé de trois modules :

- le *DES* (*Design Entry System* - le système d'entrée du projet) qui permet la spécification graphique de l'application, l'architecture cible et le placement des processus sur les processeurs. Un fichier de description du projet est généré avec une syntaxe du C parallèle;
- le simulateur *HASTE* (*HAmlet Simulator Tool (E)*) est un outil qui simule (par événements discrets) l'exécution de l'application sur une machine parallèle aussi simulée. Des traces de cette exécution sont générées;
- l'outil d'analyse de traces *TATOO*, une interface graphique utilisée pour l'analyse de traces générées par la simulation. L'utilisation des processeurs et de liens peut être visualisée.

Le *DES* supporte la modélisation de "délais" qui représentent la charge de calcul synthétique. Par exemple, étant donné un processus, son comportement peut être décrit de la façon suivante :

```
while active    /* pendant qu'il y a du travail a faire */
   picture := Receive(G1)  /* lecture d'une image prise
                              par une camera */
   picture -->area[50][50] /* garder l'image dans "area" */
   delay (10000 flop)      /* calcul modelise' */
   for i=1,50 j=1,50
```

```
      if area(i,j) = graphical
         Send (G2, area(i,j)) /* envoi de "area" si
                                 graphique */
      if area(i,j) = textual
         Send (G3, area(i,j)) /* envoi de "area" si
                                 textuel */
   end for
end while
```

La modélisation de programmes parallèles dans *HAMLET* a pour but aider la conception d'applications industrielles parallèles. A long terme, la technique développée dans ce projet sera similaire à GMB, ELGDF et ParaDiGM. Cependant, la version courante du modèle est utilisée pour la spécification de la charge de travail fournie à un simulateur. Dans ce modèle, la représentation des coûts de calcul est possible (par le moyen des instructions du type `delay`). Comme *MIMD*, les programmes modélisés suivent un paradigme de programmation très spécifique (processus séquentiels qui communiquent par le moyen de messages).

## 2.7 Conclusions

Les méthodologies de modélisation d'algorithmes présentées couvrent la majorité des approches trouvées dans la littérature. Les algorithmes parallèles sont composés de deux aspects : un flot de contrôle et un flot de données. Souvent, un des ces flots n'est pas représenté explicitement, mais il existe quand-même. Chaque méthodologie présentée permet la modélisation l'un ou les deux de ces deux aspects, insistant sur un ou autre, selon l'objectif de la modélisation.

Les graphes de précédence et de communication sont adéquats pour la modélisation des opérations et des communications de données entre ces opérations d'un algorithme parallèle. Avec ces graphes, on insiste sur la représentation du flot de contrôle. En fait, les graphes de communication, utilisés dans les études du placement, peuvent être convertis en un graphe de précédence si on sait exactement l'instant auquel la communication a lieu dans une activité. Par exemple, on considère un graphe de communication à deux tâches présenté dans la Figure 2.13. On considère encore que la communication entre les tâches $A$ et $B$ intervient respectivement après $x$ et $y$ opérations (Figure 2.13). Le graphe de l'exemple peut être transformé dans un graphe de précédence (Figure 2.14). Dans ce contexte, une communication est considérée comme une dépendance de données entre les activités.

L'introduction des probabilités et de logiques d'entrée et de sortie permet la modélisation de boucles et de tests, structures de contrôle très utilisées dans les algorithmes. Eventuellement, une communication inter-tâche (flot de donnée) est associée à une précédence (flot de contrôle). Ce flot de donnée est modélisé par une quantité. Une description explicite du flot de données d'un algorithme a été introduite par GMB,

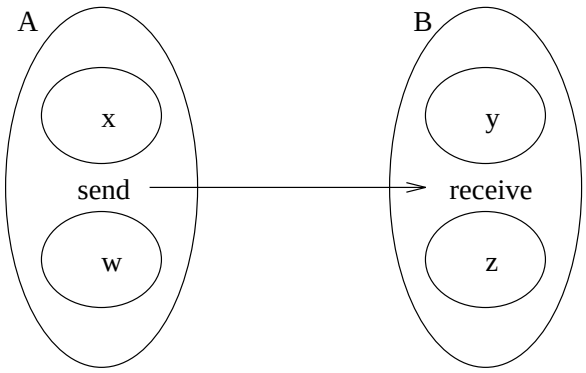



Figure 2.13 : Graphe de communication à deux tâches.

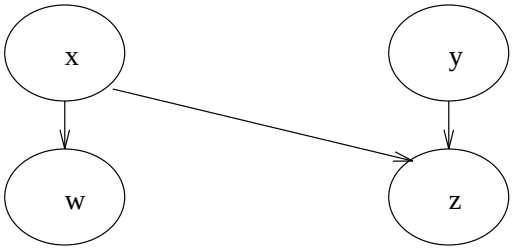



Figure 2.14 : Graphe de précédence obtenu à partir d'un graphe de communication.

ELGDF et par ParaDiGM. Les réseaux de Petri sont un outil plus abstrait. Les atouts de ce type de réseaux sont la possibilité de valider formellement le modèle (détecter, par exemple, l'occurrence probable d'une situation d'interblocage) et la disponibilité de plusieurs outils automatiques d'analyse (e.g., GreatSPN [Chi92]). Dans toutes ces techniques, on peut envisager la construction de modèles hiérarchiques. Les files d'attente se placent au même niveau de contexte d'utilisation que les réseaux de Petri, mais dans les files, il y a une claire distinction entre la modélisation du programme et la modélisation de la machine.

Le contexte de l'utilisation de chaque méthodologie est variable. Les graphes de précédence ainsi que les hypergraphes et les graphes de communication sont souvent utilisés comme des modèles d'algorithmes pour les études théoriques d'ordonnancement et de placement. D'un autre côté, GMB et les réseaux de Petri sont des outils pour la modélisation du comportement logique de systèmes qui ne sont pas forcément des systèmes informatiques. Par contre, GMB et les réseaux de Petri sont des outils plus "lourds" : des outils qui supportent la modélisation d'un plus grand nombre de comportements. A partir d'un modèle GMB ou réseau de Petri, des études théoriques

peuvent être faites et aussi qu'une vérification formelle (e.g., détection d'interblocages) et qu'une simulation. ELGDF et ParaDiGM restent au milieu. ELGDF permet la modélisation d'algorithmes parallèles dans un environnement de programmation parallèle (PPSE, un environnement de support à la conception, à l'implantation et à l'évaluation des performances de programmes parallèles). ParaDiGM est utilisé plutôt comme un outil de modélisation pour l'évaluation qualitative de différentes stratégies d'implémentation de programmes parallèles. Cette évaluation est faite, au départ, en analysant une représentation statique de l'algorithme. Aucune implantation du modèle n'est réalisée.

Les algorithmes modélisés avec ces techniques suivent différents modèles de calcul. Les graphes de précédence modélisent un flot de contrôle. En général, ce graphe n'est pas associé à un modèle de calcul particulier, quoique les graphes de tâches soient souvent très adaptés pour la modélisation de tâches séquentielles qui communiquent par des messages. Les graphes GMB et les réseaux de Petri peuvent être aussi utilisés pour la modélisation de différents modèles de calcul et de programmation : cela dépend de la sémantique donnée aux primitives de modélisation. Dans ce chapitre, les exemples présentés sont des modèles d'algorithmes composés de tâches qui peuvent communiquer par échange de messages ou par partage de mémoire. ELGDF est spécifique de la modélisation d'échange de messages et de partage de mémoire. ParaDiGM est plus limité : les algorithmes modélisés sont exécutés par des multiprocesseurs à mémoire locale ayant l'échange de messages comme mécanisme de communication entre processus.

Très peu de travaux quantifient le coût dépensé dans la construction des modèles. Deux possibilités existent :

1. à partir du modèle, on construit le programme;

2. le modèle est développé à partir d'un vrai programme.

On trouve la première approche, en général, dans les outils d'aide à la programmation parallèle comme ELGDF dans PPSE. Au départ, l'utilisateur définit un graphe (le modèle) et, avec l'aide de l'ordinateur, peut réaliser des analyses des performances, de la validation formelle, du placement, etc…sur le modèle. Lorsqu'une implémentation correcte est choisie, le code de l'application est alors ajouté au modèle. Eventuellement, l'outil peut fournir à l'utilisateur un ensemble de structures régulières (e.g, des itérations, des grilles, des arbres) préparées à l'avance, si l'application à développer fait partie d'une des ces "familles" d'algorithmes. La deuxième approche peut être trouvée dans les études d'ordonnancement de programmes parallèles.

Par rapport à l'exactitude des informations représentées dans le modèle, en général, cela dépend de l'utilisateur. S'il connaît bien l'application à modéliser, alors la représentation de la structure logique et de la dépendance des données est, en général, directe et mécanique. Par contre, extraire cette structure et cette dépendance d'un programme réel est beaucoup plus compliqué. Pour les deux cas, le grand problème reste la quantification de certains aspects, comme, par exemple, la probabilité de branchement ou

combien de fois une boucle est exécutée, si les données et leurs contenus ne sont pas représentés. En général, ces aspects dépendent des données d'entrées. Si ces données ne sont pas représentées, on utilise des approximations, des bornes ou des moyennes.

Deux aspects, présents dans les algorithmes parallèles, qui sont importants et qui sont rarement modélisés sont les entrées et sorties et les instructions temporisées du type "ne rien faire pendant 5 microseconds". Puisque les algorithmes parallèles analysés dans cette thèse ne prennent pas en compte les entrées et les sorties (plutôt le noyau de l'algorithme) et que les instructions temporisées ne sont pas utilisées souvent, nous leur donnerons une importance mineure. Quelques indications sur cette modélisation sont données dans [dJ93] où les entrées et sorties sont modélisées par des hypernœuds du type *get* et *put*. Les délais sont modélisés dans les langages orientés au traitement de signaux, comme Signal [LM90]. Les Tableau 2.1 et 2.2 présentent un sommaire des méthodologies présentées.

Finalement, nous avons fait une enquête dans le forum `comp.parallel` de *USENET news* sur la modélisation quantitative d'algorithmes. Dans toutes les réponses reçues (à peu près une dizaine), la modélisation est faite en utilisant une des techniques présentées dans ce chapitre.

| | précédence | logiques booléennes | probabilités | hiérarchie |
|---|---|---|---|---|
| graphes de précédence | √ | | | |
| graphes de communication | | | | |
| Gillies & Liu | √ | √ | | |
| Towsley | √ | | √ | |
| Kapelnikov et al. | √ | √ | √ | √ |
| Gelenbe | √ | | | √ |
| Hypergraphes | √ | | | |
| GMB | √ | √ | | |
| ELGDF | √ | | | √ |
| ParaDiGM | √ | | | |
| Petri stochastique | √ | | √ | √ |
| Files d'attente | | | √ | √ |
| MIMD | √ | | | |
| HAMLET | √ | | | |

| | communications globales | mémoire partagée | coûts | meta-structures |
|---|---|---|---|---|
| graphes de précédence | | | √ | |
| graphes de communication | | | √ | |
| Gillies & Liu | | | √ | |
| Towsley | | | √ | |
| Kapelnikov et al. | | | √ | |
| Gelenbe | | | √ | |
| Hypergraphes | | | √ | |
| GMB | | √ | | |
| ELGDF | | √ | | √ |
| ParaDiGM | √ | √ | | |
| Petri stochastique | | | √ | |
| Files d'attente | | | √ | |
| MIMD | | | √ | |
| HAMLET | | | √ | |

Tableau 2.1 : Sommaire des techniques présentées.

| | objectif |
|---|---|
| graphes de précédence | ordonnancement de tâches<br>modélisation du flot de contrôle |
| graphes de communication | placement de tâches |
| Gillies & Liu | ordonnancement de tâches |
| Towsley | ordonnancement et placement de tâches |
| Kapelnikov et al. | modélisation de la charge pour un réseau de files d'attente |
| Gelenbe | modélisation de la charge pour un réseau de files d'attente |
| Hypergraphes | modélisation de systèmes,<br>ordonnancement de tâches |
| GMB | aide à la conception, à la modélisation,<br>à la simulation et à l'analyse de systèmes |
| ELGDF | environnement de programmation parallèle |
| ParaDiGM | évaluation qualitative de programmes parallèles |
| Petri stochastique | modélisation quantitative de systèmes<br>vérification formelle/performances |
| Files d'attente | modélisation quantitative de systèmes<br>performances |
| MIMD | simulation |
| HAMLET | simulation (à long terme : conception) |

Tableau 2.2 : Sommaire des techniques présentées (continuation).

# Chapitre 3

# Le Modèle *ANDES*

Ce chapitre est consacré à la présentation du modèle *ANDES*. Ce modèle permet la représentation quantitative d'algorithmes parallèles. Les modèles générés doivent être utilisés pour **l'évaluation des performances** des aspects liés au parallélisme. En comparant avec les approches vues dans le chapitre 2, *ANDES* se veut une synthèse des approches qui se focalisent sur la description du flot de contrôle (Tableau 3.1).

| | précédence | logiques booléennes | probabilités | hiérarchie |
|---|---|---|---|---|
| *ANDES* | $\surd$ | $\surd$ | $\surd$ | $\surd$ |

| | communications globales | mémoire partagée | coûts | meta-structures |
|---|---|---|---|---|
| *ANDES* | $\surd$ | $\surd$ | $\surd$ | $\surd$ |

| | objectif |
|---|---|
| *ANDES* | utilisation pour l'évaluation des performances |

Tableau 3.1 : Sommaire des caractéristiques de *ANDES*.

Des indications de coût permettent la modélisation quantitative. La précédence, les logiques booléennes, les probabilités, les communications globales et la mémoire partagée sont associées à la modélisation de la structure de l'algorithme, du flot de contrôle et du flot de données associé. Enfin, les possibilités de description hiérarchique et par des meta-structures permettent une modélisation systématique et organisée des algorithmes parallèles.

Un modèle *ANDES* peut avoir une expression textuelle (voir le chapitre 4) ou graphique. Il peut représenter un programme organisé selon divers modèles de pro-

grammation : processus communicants, procédures appelées à distance, parallélisme de données, etc…. Les exemples présentés dans le chapitre 4 tentent d'illustrer ceci.

## 3.1 Vue d'ensemble

*ANDES* n'est pas lié à un modèle de programmation spécifique, mais plutôt à un modèle de calcul abstrait fondé sur le **parallélisme de contrôle**. Un algorithme est un ensemble d'activités qui peuvent être ou pas exécutées simultanément. Ces activités suivent un ordre de précédence dicté ou non par une dépendance de données et par la structure logique de l'algorithme. A ce niveau d'abstraction, nous ne définissons pas, par exemple, comment les données sont passées d'une activité à une autre.

Comme on l'a déjà vu, les graphes de précédence sont adéquats pour la modélisation d'algorithmes à flots de contrôle parallèles. Les sommets modélisent les activités et les arcs, la précédence dictée par une dépendance de données. Les coûts associés aux activités et aux données sont aussi importants, vu que notre intérêt va vers une approche quantitative de la modélisation. Les logiques d'entrée et de sortie permettent une "relaxation" du concept de précédence, par exemple, en permettant qu'une activité avec plusieurs prédécesseurs démarre seulement après que l'un de ses prédécesseurs soit fini.

Les graphes de précédence et de communication sont très abstraits et sont utilisés par GMB, ELGDF et ParaDiGM avec des sémantiques très différentes. Ils modélisent l'échange de messages, le partage d'un espace mémoire et même d'autres aspects comme la création de processus. On peut avoir des arcs spécifiques pour la modélisation du flot de contrôle et des arcs spécifiques pour la modélisation du flot de données. Dans *ANDES* on se focalisera dans un premier temps sur la modélisation du flot de contrôle parallèle. Une extension prévue permettra la modélisation de données partagées comme dans ELGDF.

ELGDF et ParaDiGM sont des techniques liées à la modélisation de programmes qui suivent un modèle de programmation très spécifique (par exemple, les modèles à échange de messages ou à mémoire partagée). *ANDES* prend un modèle plus abstrait où les activités ont besoin des données utilisées par d'autres activités. Ces activités peuvent échanger ces données soit par le moyen de messages, soit par une mémoire partagée, mais ce choix d'implantation n'est pas représenté dans un modèle *ANDES*. Ainsi, *ANDES* reste à mi-chemin entre les graphes de précédence et les techniques ELGDF, GMB et ParaDiGM. En fait, *ANDES* peut être considéré comme un graphe du style Kapelnikov et al. [KME89], auquel on a ajouté des aspects, par exemple, comme la possibilité de modélisation de communications globales.

*ANDES* doit, de cette façon, permettre la modélisation :

- de la précédence;
- des coûts de calcul;

- des coûts de dépendance de données;
- des comportements booléens avec ou sans utilisation de probabilités;
- des mouvements globaux de données;
- des structures hiérarchiques;
- des structures de contrôle de base;
- des envois de message;
- des données partagées (extension future).

La **modélisation de la précédence** est faite dans toutes les techniques analysées auparavant. Pour évaluer les performances des algorithmes parallèles, il faut aussi connaître l'ordre d'exécution des différentes parties de cet algorithme, car cet ordre a une influence capitale dans l'utilisation des ressources de la machine parallèle cible. La **modélisation du calcul** est aussi fondamentale. Toutes les techniques vues précédemment prévoient une modélisation adéquate d'un calcul. Dans *ANDES*, les coûts associés à chacun de ces calculs sont des variables aléatoires (éventuellement des distributions constantes). Avec ces variables aléatoires, on peut modéliser des activités contenant, par exemple, des boucles dont le nombre d'itérations n'est pas connu ou des décisions dont les branches ont un nombre différent d'opérations. *ANDES* doit aussi permettre la représentation de coûts qui sont dépendants d'autres attributs du modèle comme, par exemple, la taille d'une donnée d'entrée ou le coût de calcul précédent. Cela représente les calculs qui travaillent, par exemple, sur un vecteur. Plus le vecteur est grand, plus les calculs réalisés sont importants. La **dépendance des données** est aussi modélisée de façon quantitative. Le coût correspond à la taille de la donnée communiquée d'une activité à autre. Comme pour le calcul, ce coût suit une distribution aléatoire (éventuellement une distribution constante) et peut être dépendant d'autre attribut du modèle. Par exemple, la taille d'une donnée peut dépendre du nombre d'opérations de l'activité qui produit cette donnée. La modélisation des différents comportements de communication se fait principalement en utilisant **les logiques booléennes et les probabilités**. Les logiques booléennes indiquent quelles communications doivent avoir lieu. Les probabilités permettent la modélisation d'un flot de contrôle qui dépend des données (par exemple, une décision ou les boucles). La modélisation d'une **communication globale** permet la modélisation de plusieurs activités qui dépendent d'une même donnée en même temps. ParaDiGM est un modèle qui prévoit ce type de comportement en permettant la modélisation d'une diffusion d'une donnée. Enfin, l'utilisation de **descriptions hiérarchiques** permet le raffinement systématique de modèles et une possible évaluation à différents grains de calcul. Les **structures de base** utilisées dans la modélisation d'algorithmes (étant souvent régulières) simplifient la construction des modèles. Par exemple, au lieu de modéliser chaque activité d'un modèle ayant une structure de grille, on peut décrire une seule activité et la structure d'interdépendance

entre les diverses copies de cette activité (bref, la description de la grille). C'est le rôle joué en partie par les "meta-structures" de ELGDF. Enfin, deux aspects souvent présents dans les algorithmes parallèles sont les **envois de message** et l'accès à une **donnée partagée** par plusieurs flots de contrôle. Lorsque les communications sont associées aux précédences, l'échange de données par messages a lieu lorsqu'un calcul, modélisé par un sommet du graphe, se termine. D'autre part, le conflit d'accès à une donnée partagée dépend des vitesses des exécutions des flots de contrôle qui utilisent cette donnée. Le conflit induit des attentes et influe sur les performances du système. Ainsi, la modélisation de ces données est importante. GMB et ELGDF sont des techniques qui supportent la représentation de ces entités. Le traitement des données partagées n'est pas considéré dans la version actuelle de *ANDES*.

*ANDES* ne contient pas de primitives particulières pour la modélisation de certains aspects dynamiques, comme la création de processus (voir les ellipses employées dans ParaDiGM – chapitre 2). Dans *ANDES*, si la modélisation d'une activité "création de processus" est nécessaire, alors cette activité est modélisée comme un calcul banalisé dont le coût est déterminé par la machine cible d'exécution. De même, les entrées et sorties peuvent être représentées par des activités banalisés avec des annotations du type "trois entiers" pour un coût de lecture ou "un flottant et une chaîne de 20 caractères" pour un coût d'écriture. Les opérations dépendantes du temps comme "attendre cinq microsecondes" peuvent être modélisées de façon analogue. "Attendre" est l'opération. Cinq microsecondes est le coût.

## 3.2 Le modèle *ANDES*

Nous détaillons ici le modèle *ANDES*. Un modèle *ANDES* est un graphe orienté sans circuit (en anglais *DAG = Directed Acyclic Graph*) annoté où les sommets modélisent les **nœuds de calcul** (c'est-à-dire, les activités) et les arcs la **précédence** entre ces nœuds de calcul. Un graphe *ANDES* s'appelle *DG-ANDES* (*DG = Directed Graph*). Un nœud de calcul est composé d'une **annotation d'entrée**, d'une **annotation de sortie** et d'une **annotation de calcul**. Les annotations d'entrée et de sortie détaillent les dépendances existantes entre les nœuds de calcul. Ces dépendances sont associées aux arcs de précédence. L'annotation de calcul modélise le traitement de données. Dans une annotation de calcul, on peut avoir soit un traitement simple considéré comme une suite d'instructions, soit un traitement plus complexe, modélisé par un autre graphe *ANDES*. C'est grâce à cette possibilité d'avoir un *DG-ANDES* dans une annotation de calcul que l'on peut construire des modèles hiérarchiques et récursifs.

Un nœud de calcul a un identificateur. Lorsqu'on veut se référer aux annotations d'un nœud de calcul ayant $X$ comme identificateur, $LE(X)$, $LC(X)$ et $LS(X)$ identifient respectivement l'annotation d'entrée, l'annotation de calcul et l'annotation de sortie du nœud de calcul $X$. La Figure 3.1 présente un exemple d'un nœud de calcul.

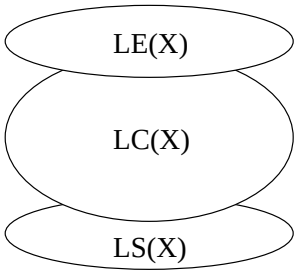



Figure 3.1 : Nœud de calcul.

### 3.2.1 Annotation de calcul

Les annotations de calcul précisent la quantité de calcul réalisé dans un algorithme parallèle. Une annotation de calcul a trois attributs :

- **un identificateur**;
- **un coût**;
- **un *DG-ANDES*** associé (éventuellement).

L'**identificateur** est donné par $LC(X)$ où $X$ est l'identificateur du nœud de calcul. Le coût représente une quantité d'opérations de base devant être réalisée par le calcul et est exprimé par une fonction numérique. Le type d'opération de base est laissé au choix du modélisateur (e.g., opérations scalaires, flottantes, binaires, etc…). Un exemple de cette démarche est donné par le Lawrence Livermore National Laboratory [LER92] qui a choisi de ne compter que le nombre d'opérations virgule flottante dans un algorithme (Figure 3.2).

Différent types de fonctions sont utilisés pour représenter le coût d'une annotation de calcul. Ces possibilités sont décrites du plus simple au plus complexe :

- une **constante** lorsqu'on modélise un calcul avec une séquence fixe d'opérations travaillant sur un ensemble de données constantes (soit en valeur, soit en taille) et connu en avance (Figure 3.2);
- une **distribution aléatoire** lorsqu'on modélise un calcul non déterministe, c'est-à-dire dont le nombre d'opérations n'est pas connu à l'avance. L'utilisation de distributions aléatoires modélise l'influence des données sur un algorithme étant donné que les noms et les contenus des structures de données sont absents de notre description. Par exemple, le coût d'un calcul en boucle peut être représenté par une distribution aléatoire, dont on ne connaît pas *a priori* le nombre d'itérations (Figure 3.3). Il est vrai que l'obtention de cette distribution n'est

| *Opérations* | *Poids* |
|---|---|
| $a = b$, $a =$(constante) | 0 |
| $a < 0$, $a <= 0$, $a == 0$, $a! = 0$, $a > 0$, $a >= 0$ | 0 |
| $-a$, fabs($a$), fsgn($a$,$b$) | 0 |
| $a + b$, $a - b$, $a * b$, $a^2$ | 1 |
| $a < b$, $a <= b$, $a == b$, $a! = b$, $a > b$, $a >= b$ | 1 |
| (int)$a$, (double)$b$ | 1 |
| $1/a$, $-1/a$ | 3 |
| $a/b$ | 4 |
| sqrt($a$) | 4 |
| conversion de ou vers une chaîne ASCII | 6 |
| sin($a$), cos($a$), tan($a$), log($a$), atan($a$), exp($a$) | 8 |

Tableau 3.2 : Exemple de choix pour le poids des opérations.

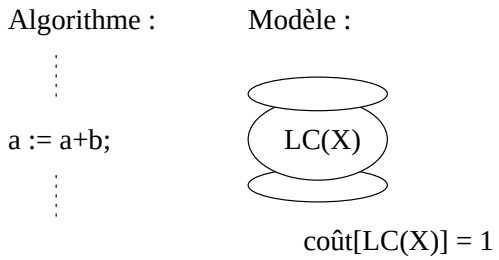



Figure 3.2 : Coût constant d'une annotation de calcul.

pas évidente, mais en connaissant les valeurs possibles maximales et minimales qu'une variable peut contenir, des distributions représentatives de la moyenne, des intervalles de variation, de la variance peuvent être utilisées;

- une **fonction dont les paramètres sont d'autres attributs du *DG-ANDES***. Avec ces fonctions, nous trouvons une autre façon de modéliser les données. Les paramètres de ces fonctions sont d'autres coûts du graphe ou bien des paramètres du modèle lui-même (la notion de "paramètre du modèle" correspond au désir de modéliser un calcul en fonction de la taille des entrées de l'algorithme modélisé - par exemple, une multiplication de matrices de taille $N$, $N$ étant le paramètre du modèle). Par exemple, supposons qu'un certain calcul $X$ est effectué sur des données qui arrivent dans un message. Si le nombre d'opérations de $X$ dépend du contenu du message reçu, ce nombre peut être modélisé par une distribution aléatoire. Mais, ce nombre d'opérations peut aussi dépendre de la

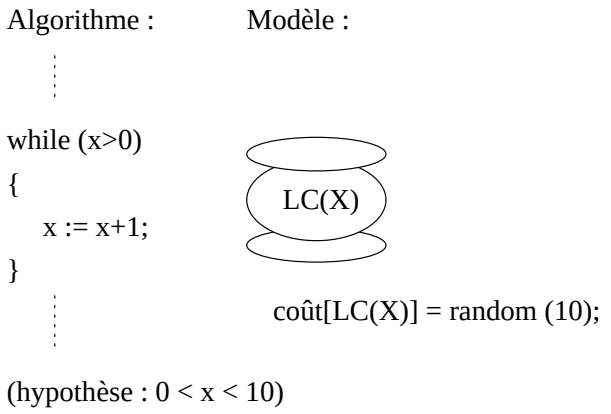



Figure 3.3 : Coût aléatoire d'une annotation de calcul.

**taille** du message reçu (en fonction du nombre, par exemple, d'entiers ou de réels). On verra plus tard dans ce chapitre que la taille d'un message est un coût de communication. Dans la modélisation du calcul $X$ ci-dessus, si le nombre d'opérations de $X$ dépend seulement de la taille du message reçu (Figure 3.4), alors on a un exemple de coût de calcul dépendant d'un autre coût du *DG-ANDES*.

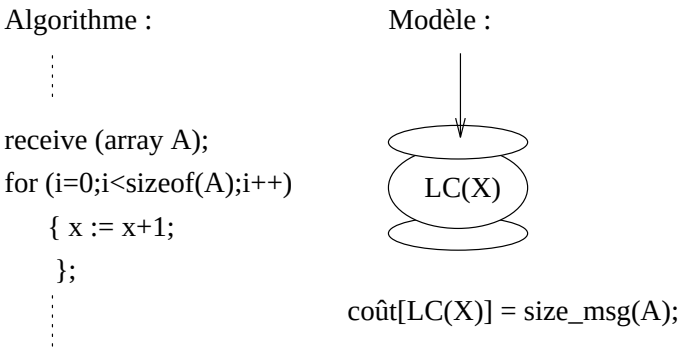



Figure 3.4 : Coût dépendant d'une annotation de calcul.

- une **fonction mixte** qui mélange des distributions aléatoires et les dépendances des coûts. Dans l'exemple précédent, si le nombre d'opérations du calcul $X$ dépend de la taille du message et aussi de son contenu, alors le coût associé à ce calcul peut être exprimé par une distribution aléatoire, dont un paramètre (par exemple, la moyenne) tient compte d'un autre coût : la taille de message.

De façon à permettre une modélisation hiérarchique, une annotation de calcul peut être éventuellement un autre *DG-ANDES*. Une discussion sur les descriptions hiérarchiques est présentée plus tard dans ce chapitre (paragraphe 3.2.3). Nous verrons que la hiérarchie peut être utile pour une étude de granularité de calcul et pour la représentation d'algorithmes récursifs. Par contre, nous verrons aussi que la hiérarchie impose certaines contraintes au modèle de programmation modélisé.

### 3.2.2 Annotations d'entrée et de sortie

Les annotations d'entrée et de sortie d'un nœud de calcul permettent de modéliser les dépendances de données entre les nœuds de calcul du modèle. Ces dépendances sont associées aux relations de précédence. Le terme "annotation d'entrée" vient du fait que cette annotation doit être "résolue" avant l'annotation de calcul correspondante. De même, le terme "annotation de sortie" vient du fait que cette annotation est "résolue" après le calcul correspondant. Un nœud de calcul qui n'a pas de prédécesseur est dit **racine** et n'a pas d'annotation d'entrée. Un nœud de calcul qui n'a pas de successeur est dit **feuille** et n'a pas d'annotation de sortie (Figure 3.5).

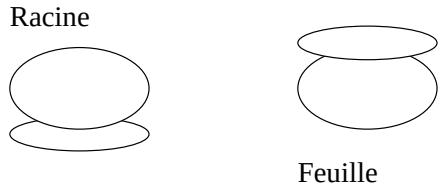



Figure 3.5 : Nœuds de calcul racine et feuille.

Les annotations d'entrée et de sortie précisent la quantité de communication qui a lieu entre les nœuds de calcul du *DG-ANDES*. Une annotation d'entrée/sortie a trois attributs :

- **un identificateur**;
- **un coût**;
- **un descripteur de type**.

L'**identificateur** d'une annotation d'entrée est donné par $LE(X)$, où $X$ est l'identificateur du nœud de calcul. De même, l'**identificateur** d'une annotation de sortie est donné par $LS(X)$.

Comme pour le calcul, un **coût** est associé aux annotations d'entrée et de sortie. Nous insistons sur le fait que ces coûts sont liés aux caractéristiques de l'algorithme. Ils ne peuvent pas être liés, à ce niveau de description, au matériel et aux logiciels

sous-jacents. Dans un *DG-ANDES*, le coût d'une dépendance correspond à la taille de la donnée par rapport à une unité de base (e.g., un entier, un caractère). Comme pour le calcul, ce coût peut être une constante, une variable aléatoire ou une valeur dépendante d'un autre attribut du graphe (par exemple, une structure de donnée dont la taille dépend du nombre d'opérations qui produit cette donnée).

Les **descripteurs de types** d'entrée et de sortie peuvent être :

- simples;
- une annotation booléenne;
- une opération globale.

Un descripteur d'entrée/sortie **simple** correspond à une dépendance de données entre deux nœuds de calcul (Figure 3.6).

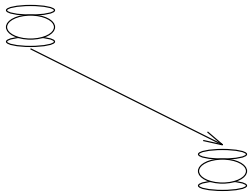


Figure 3.6 : Annotation simple d'entrée et de sortie.

Les **annotations booléennes** suivent l'approche de Kapelnikov et al. (voir le chapitre 2). Le but est de définir, par le moyen d'expressions booléennes, un comportement d'acceptation ou d'envoi de communications entre les nœuds de calcul. Pour toute communication, on peut avoir :

- une annotation `AND` : elle correspond au cas où toutes communications doivent avoir lieu avant le début du calcul (Figure 3.7);

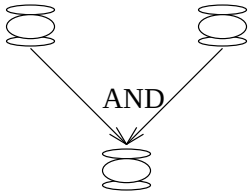



Figure 3.7 : Annotation d'entrée du type `AND`.

- une annotation OR : la première communication qui arrive est acceptée. Si plusieurs communications arrivent en même temps, une est choisie au hasard. Cette structure modélise, par exemple, l'instruction ALT d'Occam2 [Lim88] (Figure 3.8).

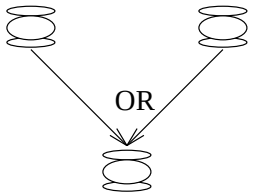



Figure 3.8 : Annotation d'entrée du type OR.

Les annotations de sortie acceptables pour l'échange de messages sont :

- l'annotation AND qui permet de modéliser l'envoi en parallèle de plusieurs communications (Figure 3.9);

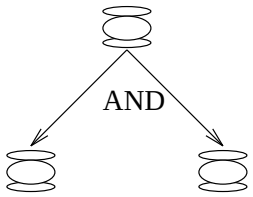



Figure 3.9 : Annotation de sortie du type AND.

- l'annotation OR est composée de trois types : l'annotation OR probabiliste, l'annotation OR déterministe et l'annotation OR indépendant. La annotation OR probabiliste (OR-PROB) consiste à activer une sortie selon une distribution discrète du type Bernoulli multiple. Chaque sortie $i$ a une probabilité associée $p_i$ et $\sum p_i = 1$. Seulement une sortie est choisie. Cette annotation modélise les décisions dans un algorithme (Figure 3.10). L'annotation OR déterministe (OR-DET) permet de modéliser un envoi sélectif de communications selon le résultat de l'annotation d'entrée du type OR du même nœud. Par exemple, dans la Figure 3.11, la porte $0$ de $LS(A)$ est activée si un message est arrivé par la porte $0$ de $LE(A)$. Si un message arrive par la porte $1$ de $LE(A)$, alors un message doit être envoyé par la porte $1$ de $LS(A)$. Cette annotation modélise, par exemple, l'envoi d'un message par un canal dont l'identificateur dépend du résultat d'une instruction ALT. Une

autre variation sur l'annotation OR peut être utilisée, comme le OR indépendant (OR-IND), où une probabilité d'émission est associée à chaque sortie et tirée indépendamment une d'autre (Figure 3.12).

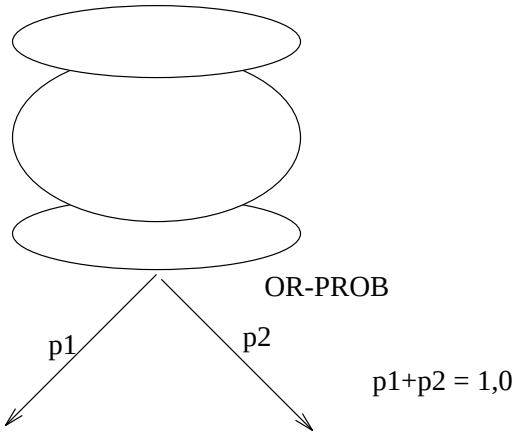



Figure 3.10 : Annotation de sortie du type OR-PROB.

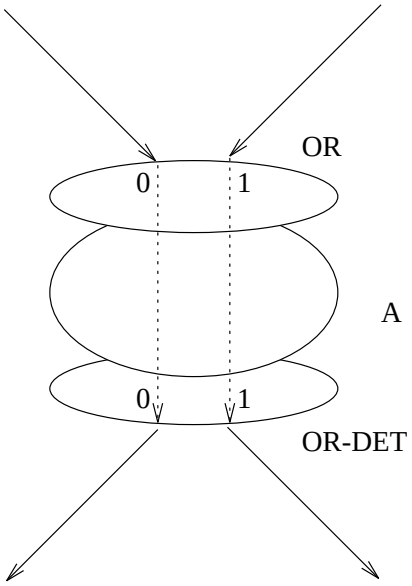



Figure 3.11 : Annotation de sortie du type OR-DET.

On doit utiliser ces annotations avec précaution, car certaines combinaisons peuvent amener l'algorithme à une situation d'interblocage. Par exemple, dans la Figure 3.13, le nœud de calcul $D$ ne s'exécute jamais.

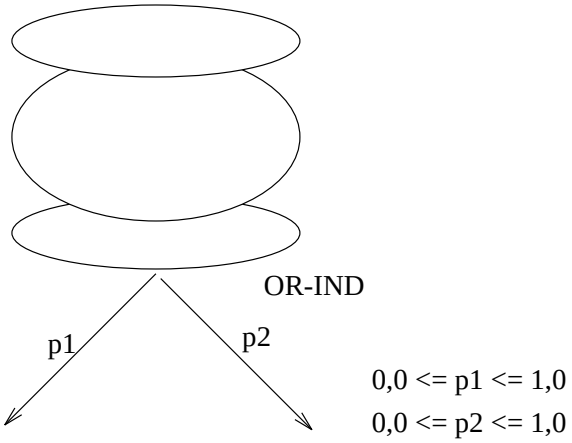



Figure 3.12 : Annotation de sortie du type `OR-IND`.

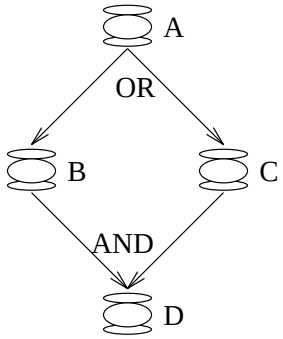



Figure 3.13 : Interblocage.

Certaines communications entre les nœuds de calcul correspondent à des diffusions ou à des regroupements d'un même ensemble de données. Nous appelons ces communications d'**opérations globales**. Ces opérations sont très communes dans les algorithmes parallèles. Il y a deux opérations de base utilisées dans la modélisation d'algorithmes (Figure 3.14) :

- `diffusion` : une donnée est envoyée par un nœud de calcul à plusieurs nœuds de calcul;
- `rassemblement` : une donnée est partitionnée sur plusieurs nœuds de calcul et recomposée sur un nœud de calcul. Cette recomposition peut être associé à un calcul (par exemple, la somme globale).

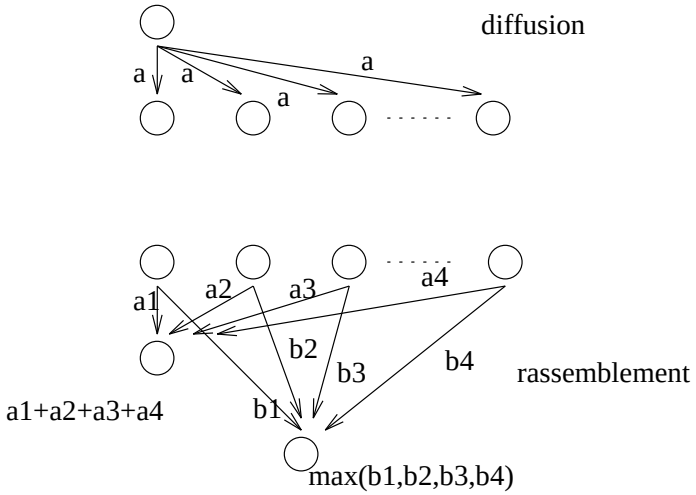



Figure 3.14 : Opérations globales.

Nous avons présenté ci-dessus les éléments de base d'un *DG-ANDES*. L'étude d'autres techniques de modélisation (voir le chapitre 2), d'algorithmes/programmes parallèles réels et de différents langages de programmation parallèle a montré que ses composants sont les plus répandues dans l'algorithmique parallèle. Par exemple, il est possible d'avoir des annotations booléennes plus complexes avec les descripteurs de base `AND` et `OR`. Cependant, tels comportements ne sont pas habituels. De même, plusieurs types de diffusion et de rassemblement existent (la diffusion elle-même, l'échange total, la distribution, la multi-distribution [T+92]), les différences étant principalement dû à l'ensemble d'émetteurs/récepteurs et au contenu du message. Dans *ANDES*, seulement la taille des messages est représentative. Ainsi, les différents types de diffusion et de rassemblement sont modélisés adéquatement par les opérations globales proposées ci-dessus. Il est important de signaler ces types d'opération dans le modèle de l'algorithme, car plusieurs architectures parallèles ont de matériel spécifique pour les réaliser de façon performante.

### 3.2.3 Hiérarchies

Plusieurs techniques de modélisation d'algorithmes présentées dans le chapitre 2 permettent la construction de **modèles hiérarchiques**. Au niveau du développement du modèle, une approche hiérarchique est souhaitable afin de systématiser ce processus de construction, car un algorithme peut être une entité très complexe. Pour l'évaluation des performances des systèmes parallèles, la hiérarchie représente différents grains de calcul. Plus on monte dans la description hiérarchique, plus gros sont les grains de calcul. Etudier les performances d'un algorithme parallèle selon différents grains de

calcul correspond à trouver un bon compromis entre la répartition de calcul et le coût de la communication.

Deux approches sont possibles pour utiliser la hiérarchie. L'approche "montante" (en anglais *bottom-up*) a, comme point de départ, un *DG-ANDES* le plus détaillé possible, c'est-à-dire, où chaque opération de base correspond à un nœud de calcul atomique. A partir de ce *DG-ANDES* très fin, on **groupe** les nœuds de calcul et on obtient un nœud de calcul à grain plus gros. Cette démarche peut être répétée jusqu'à ce que l'algorithme soit un unique nœud de calcul. Cette approche n'est pas adéquate lors de la construction du *DG-ANDES*, car son point de départ est le *DG-ANDES* à granularité la plus fine possible, c'est-à-dire, un *DG-ANDES* déjà **construit**. Par contre, lorsque le *DG-ANDES* est totalement construit, il est intéressant de grouper les nœuds de calcul pour obtenir des calculs à grain plus gros. Ainsi, un coût de communication spécial `CONTEXT` (voir le paragraphe 6.2.5) à été crée pour modéliser un groupement de nœuds de calcul. Si entre deux nœuds de calcul il existe une communication de coût `CONTEXT`, alors ces deux nœuds de calcul doivent être exécutés sur un même processeur. L'utilisation des coûts `CONTEXT` ne permet pas de modéliser plusieurs couches de groupement : les nœuds de calcul du *DG-ANDES* sont ou ne sont pas groupés.

Dans la deuxième approche, dite "descendante" (en anglais *top-down*), tout l'algorithme est vu d'abord comme un seul nœud de calcul qui peut être raffiné par un *DG-ANDES*. Comme on l'a vu, un nœud de calcul peut contenir un *DG-ANDES* comme annotation de calcul (voir paragraphe 3.2.1). A chaque niveau du raffinement, des coûts approximatifs de calcul et de communication sont utilisés, vu que le *DG-ANDES* détaillé n'est pas encore connu.

Un problème à régler est la compatibilité des annotations d'entrée et de sortie entre les différents niveaux de la hiérarchie. Pour faciliter l'expression de cette compatibilité, à chaque raffinement, un nœud d'entrée ($NE$) et de sortie ($NS$) doivent être crées. Le *DG-ANDES* qui raffine un nœud de calcul $A$ ne doit avoir qu'un seul nœud d'entrée ($NEA$) et un seul nœud de sortie ($NSA$) (Figure 3.15).

L'annotation d'entrée de $NE$ est identique à l'annotation d'entrée du nœud de calcul à raffiner (dans la Figure 3.15, $LE(A) \sim LE(NEA)$). De même, l'annotation de sortie de $NS$ est identique à l'annotation de sortie du nœud de calcul à raffiner (dans la même Figure, $LS(A) \sim LS(NSA)$). Un problème délicat de compatibilité a lieu lorsque $LS(A)$ dépend du résultat de $LE(A)$. Ceci arrive avec un $LS(A)$ du type `OR-DET` (un descripteur qui modélise, par exemple, le `ALT` d'Occam2). Dans ce cas, il faut que l'annotation de sortie de $NS$ soit dépendante des entrées de $NE$. Ce comportement peut être comparé avec la dépendance de coûts (un coût de calcul en fonction d'un autre coût du *DG-ANDES*). Il faut que $NS$ connaisse les entrées qui ont eu lieu dans $NE$ (Figure 3.16).

Un exemple d'utilisation de la hiérarchie est la description d'algorithmes récursifs. La Figure 3.17 montre une modèle de l'algorithme qui calcule la factorielle. On suppose que le nombre d'appels récursifs est contrôlée par une probabilité qui décroît à chaque fois que cet appel est exécuté.

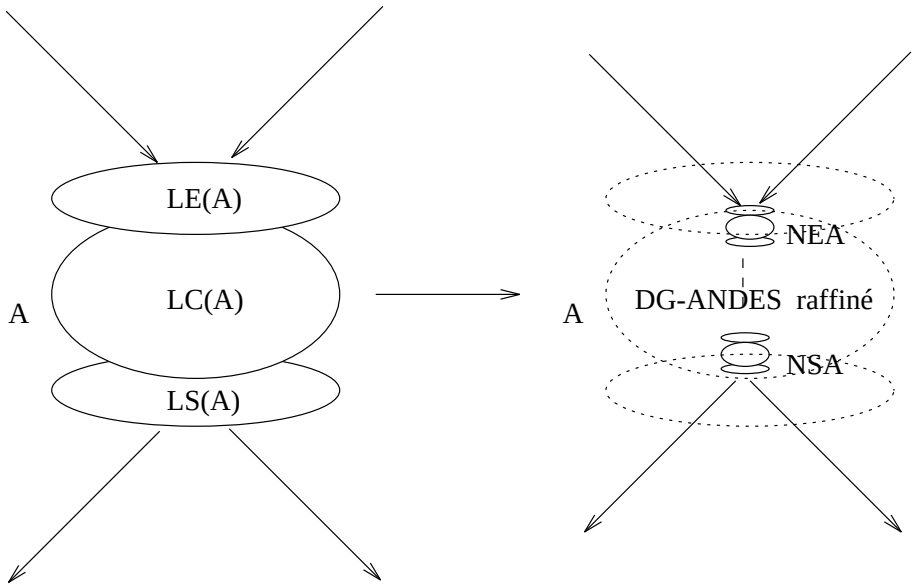



Figure 3.15 : Raffinement dans *ANDES*.

### 3.2.4 Structures de base régulières

De nombreux algorithmes du calcul intensif ont une structure régulière. Les grilles, les arbres ou les anneaux de nœuds de calcul sont très courants. *ANDES* fournit ces structures comme opérateurs de construction paramétrés (par exemple, taille de la grille). Ces structures simplifient en temps et en espace la construction des *DG-ANDES*. Par exemple, dans la Figure 3.18, une structure régulière en grille est présentée.

## 3.3 *ANDES* ne modélise pas tout…

Par rapport aux techniques présentées dans le chapitre 2, le langage *ANDES* ajoute des annotations qui permettent la modélisation quantitative pour l'évaluation des performances. Par contre, *ANDES* ne prend pas en compte quelques caractéristiques des autres techniques, plutôt associées au développement de programmes parallèles. Ces aspects des algorithmes parallèles ne sont pas modélisés, soit parce que ces caractéristiques sont très rares ou particulières à un algorithme donné, soit parce que ces aspects ne sont pas intéressants dans un contexte de modélisation quantitative. Par exemple, *ANDES* ne modélise pas explicitement l'utilisation des données. Pourtant, un minimum d'information concernant les données est représenté dans le modèle. Ce minimum est présent dans les probabilités, dans les annotations booléennes et dans les coûts aléatoires et dépendants.

*ANDES* ne modélise pas l'utilisation des données partagées. L'aspect principal lié à cette mémoire partagée est le conflit d'accès par plusieurs processus. Ce conflit peut générer des temps d'attente importants pendant l'exécution et influer sur les performances de l'exécution de l'algorithme (Figure 3.19). La modélisation de cet aspect est prévue dans une version future de *ANDES*.

*ANDES* permet la modélisation de coûts de calcul et de communication, mais le coût de l'utilisation de la mémoire ne peut pas être représenté. Ce coût est important : il sera représenté dans la prochaine version de *ANDES*.

Certaines opérations non modélisées dans *ANDES* sont très liées à la machine qui exécute le programme. Par exemple, une opération du type *reschedule* (suspendre le processus courant et le mettre à la fin de la file des processus actifs) peut être modélisée comme un nœud de calcul. Nous pouvons créer un nœud de calcul spécial nommé *reschedule*, pour modéliser cette opération.

Un autre exemple d'opération non modélisée dans *ANDES* est la communication non-bloquante. Un envoi non-bloquant consiste à mettre le message dans un tampon. Le processus émetteur n'attend pas que ce message soit reçu par le récepteur : l'émetteur continue à s'exécuter après l'envoi. Eventuellement, l'émetteur peut vérifier si le message a été reçu par le moyen d'une opération spécifique du type "consultation de la file d'envoi de messages". Une réception non-bloquante se comporte de façon analogue : on exécute l'instruction de réception. Si le message est déjà arrivé, la réception est faite. Sinon, les instructions après la réception sont exécutées. Eventuellement, le processus peut demander plus tard si le message est arrivé. Ces communications asynchrones ne sont pas prévues dans *ANDES*. Le problème est que, en utilisant ces envois et ces réceptions non-bloquantes, la précédence n'est pas connue. Dans une communication bloquante, les opérations qui suivent la réception sont exécutées forcement après les opérations antérieures à l'émission. Dans la Figure 3.20, les nœuds de calcul $A$ et $B$ communiquent de façon non-bloquante. L'arc gras modélise un envoi non-bloquant, mais pas une précédence, car $A$ peut s'exécuter avant ou après $B$. Les communications non-bloquantes sont utilisées par exemple dans les algorithmes itératifs asynchrones [BT89].

Autre aspect non détaillé dans *ANDES* est l'interaction de l'algorithme avec le monde extérieur. Les lectures de données et l'affichage des résultats peuvent être modélisées comme des nœuds de calcul spéciaux ayant comme coût la quantité de données lues ou écrites.

## 3.4 Conclusions

Comme on peut le constater en regardant les paragraphes ci-dessus, *ANDES* présente une série d'extensions au graphe orienté de base, présenté dans le chapitre 2. En effet, *ANDES* cherche à ajouter, dans une seule technique, les atouts de ces méthodes pour l'évaluation des performances.

Nous faisons dans ce paragraphe un résumé de ces extensions :

- décomposition de l'algorithme parallèle en plusieurs activités (nœuds de calcul);
- modélisation de la précédence entre ces nœuds;
- modélisation de la communication entre nœuds de calcul;
- un nœud de calcul est une entité composée d'annotations (de calcul, d'entrée et de sortie);
- utilisation d'expressions booléennes pour la modélisation de différentes dépendances entre les nœuds de calcul;
- modélisation d'opérations globales du type diffusion et rassemblement de données;
- coûts aléatoires : modélisation d'opérations et de communications qui ne sont pas bien modélisées par des coûts constants. Il s'agit d'une modélisation des données;
- coûts en fonction des attributs du graphe : modélisation, e.g., d'opérations dépendantes des tailles de communications arrivants ou dépendantes d'attributs du graphe (par exemple, la profondeur, si le graphe est un arbre);
- probabilités : modélisation de structures de contrôle basées sur les décisions (e.g., itérations).

Pour simplifier et systématiser la construction d'un *DG-ANDES*, le langage prévoit encore :

- un support pour les descriptions hiérarchiques;
- un support pour la définition des structures paramétrables représentant des graphes réguliers, e.g., des grilles et des arbres.

*ANDES*, en dépit de sa complexité, n'est pas une technique complète. Un algorithme parallèle a trois aspects de base : le contrôle, la mémoire et la communication. L'aspect mémoire n'est pas modélisé précisément dans *ANDES*. Les données ne sont pas identifiées par leurs noms, comme dans ELGDF ou GMB. Nous nous y intéressons seulement à travers leurs tailles, étant donné que la modélisation est essentiellement quantitative. De même, le coût de l'accès mémoire n'est pas pris en compte. D'autres opérations, comme les entrées et sorties, peuvent être modélisées par le moyen de nœuds de calcul spéciaux.

Ce chapitre avait pour but de présenter une méthodologie pour la modélisation d'algorithmes parallèles. Le modèle décrit la structure et le comportement de l'algorithme sans tenir compte des aspects de son implémentation. Dans le prochain chapitre, nous présentons une réalisation du langage *ANDES* qui nous permet d'exprimer le modèle *ANDES* et de l'utiliser dans un outil d'évaluation des performances. Des exemples plus détaillés sont aussi présentés.

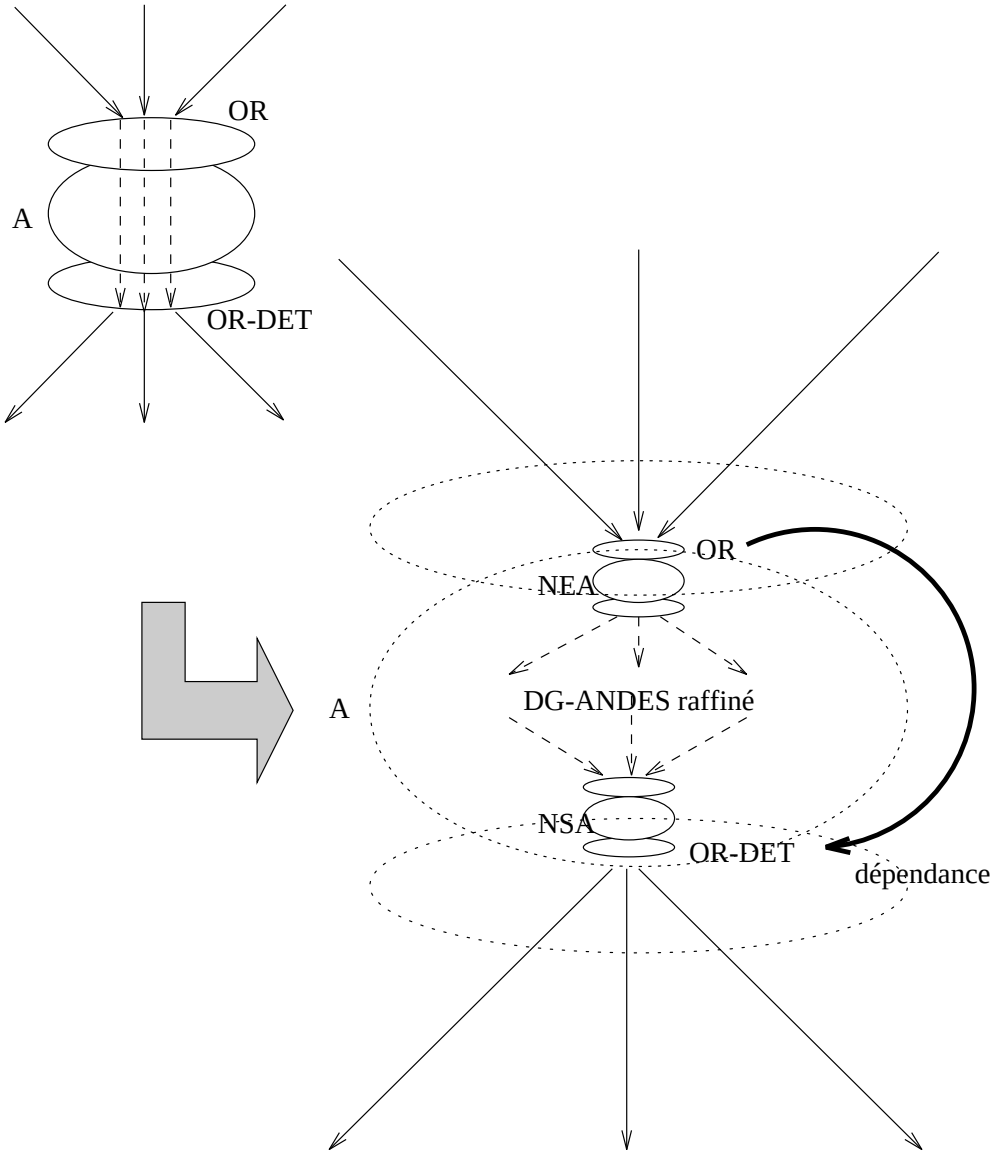
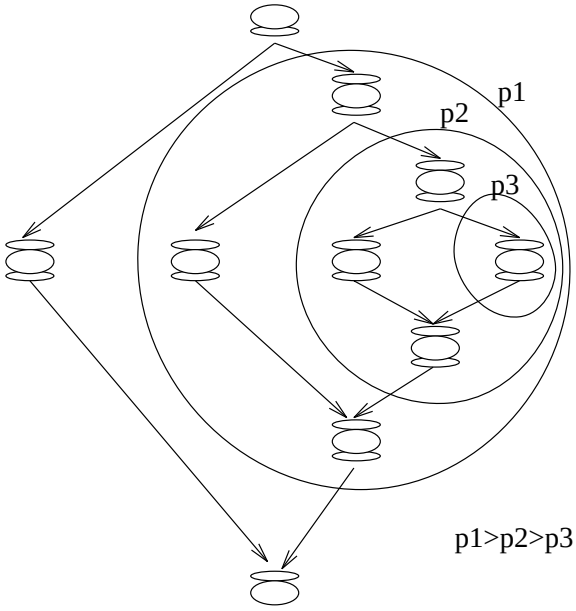



Figure 3.16 : Compatibilité entre les entrées et les sorties.

```
int factorial (x) {
    if (x == 1)
      return (1)
    else
      return (x*factorial(x-1));                % p1, p2, p3,...
}
```




pi : probabilité d'exécution

Figure 3.17 : Récursivité et hiérarchie.

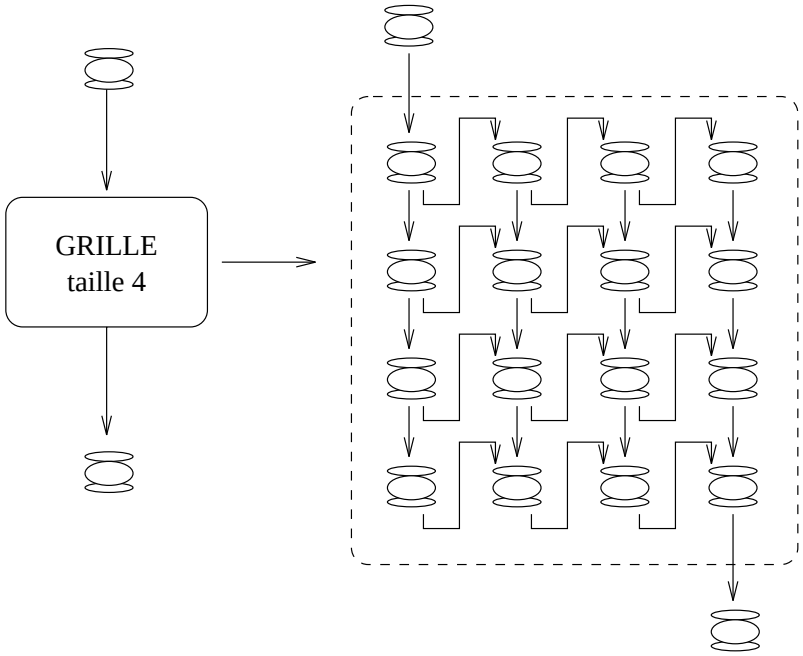



Figure 3.18 : Structure régulière.

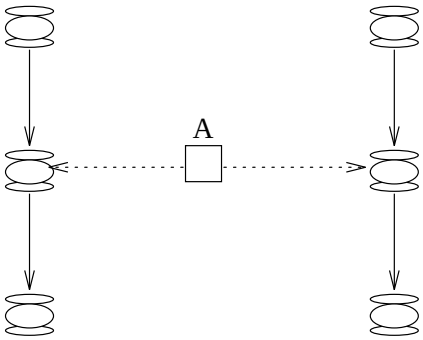



Figure 3.19 : Une donnée partagée.

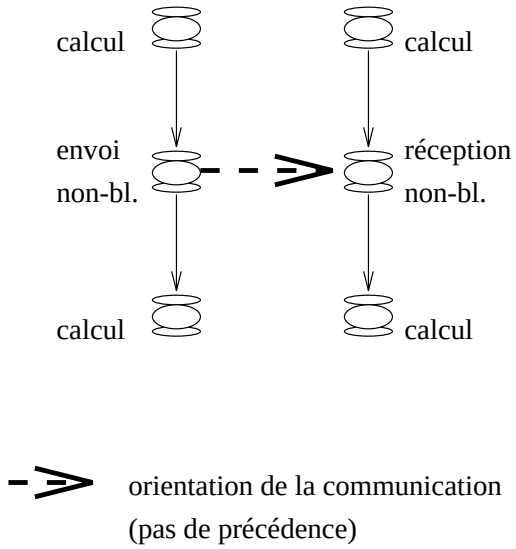



Figure 3.20 : Communication non-bloquante.

# Chapitre 4

# La Bibliothèque *ANDES-C*

Dans le chapitre 3, nous avons présenté le modèle *ANDES*. *ANDES* est une technique générique de modélisation d'algorithmes parallèles. "Générique", car *ANDES* permet la description de graphes orientés sans circuit (*DAGs*) annotés et ces graphes peuvent être utilisés dans différents environnements d'évaluation des performances. Les graphes modélisent la **charge de travail** imposée à la machine parallèle. Par exemple, Gelenbe [Gel89], Towsley [Tow86] et Baccelli [BL90] utilisent les graphes non orientés dans un environnement d'évaluation des performances à base de systèmes de files d'attente. Wabnig [WH94] utilise les graphes non orientés comme entrée des modèles de Réseaux de Petri Stochastiques et Ferscha [FJ94] les utilise comme entrée d'un simulateur à événements discrets. Dans *MIMD* [CS92], des graphes orientés valués représentent la charge de travail donnée aussi à un simulateur. On verra dans le chapitre 5 que *ANDES* est utilisé pour modéliser une charge de travail dans un environnement utilisant les programmes synthétiques.

*ANDES* permet la modélisation d'algorithmes parallèles, indépendamment d'où et de comment cet algorithme est implémenté. Pour être utilisable, le principe de **description** d'un modèle *ANDES* doit être simple. Nous avons choisi un langage de programmation hôte (le langage C séquentiel) pour décrire de modèles *ANDES*. Grâce à une bibliothèque spécifique nommé `andes.h`, une description *ANDES* est un programme C. Nous profitons de la possibilité du langage hôte d'utiliser les structures de contrôle itératives. Nous profitons aussi de la possibilité d'utiliser des procédures et des fonctions.

Nous avons intentionnellement écarté les interfaces graphiques pour décrire les modèles *ANDES* pour deux raisons. Tout d'abord, leur pouvoir d'expression et d'abstraction pour de gros modèles peut être mis en cause. Il n'y a pas dans la littérature des méthodes largement acceptables pour la description et pour la visualisation de modèles comportant des milliers de nœuds de calcul. Ensuite, si on le désire, il sera raisonnablement facile d'adapter un éditeur graphique au dessus de l'interface de programmation présentée ici.

Ainsi, dans ce chapitre, nous présentons la bibliothèque *ANDES* et les nouvelles fonctions C (définies dans cette bibliothèque) permettant de décrire des modèles *AN-*

*DES*. Ce mode d'expression est appelé informellement *ANDES-C*. Quelques exemples plus complexes de modèles *ANDES* sont aussi présentés dans ce chapitre. *ANDES-C* est surtout **le langage C plus une bibliothèque** `andes.h`. Cette bibliothèque définit une interface d'utilisation qui permet la construction de modèles *ANDES*.

Suivant le contenu de la bibliothèque `andes.h`, la compilation et l'exécution d'une description *ANDES-C* peut produire un *DG-ANDES* dans un format interne afin d'être utilisé, par exemple, par un simulateur ou par un générateur de charge synthétique. Cette exécution peut aussi produire directement un programme synthétique ou un squelette d'un programme parallèle modélisé. Ce squelette pourra être utilisé postérieurement dans l'écriture du programme parallèle réel.

## 4.1 La structure d'un modèle *ANDES-C*

Une description d'un modèle *ANDES* est un programme C ayant le format suivant :

```
#include <andes.h>  /* inclusion de la bibliotheque */

/* partie A : declaration des noeuds de calcul */

/* partie B : definition des annotations d'entree, */
/* de sortie et de calcul                          */

int main () {

   /* partie C :                       */
   /* definition des noeuds de calcul */

   /* partie D :                       */
   /* definition des precedences       */
}
```

Les objets de base d'un modèle *ANDES* sont les nœuds de calcul. La bibliothèque `andes.h` contient la définition d'un type "nœud-de-calcul" (le type `Node` : voir paragraphe suivant) et la définition des nouvelles méthodes (des fonctions écrites en langage C) de manipulation des objets de ce type. Dans la partie A du format présenté ci-dessus, les nœuds de calcul sont déclarés. Dans la partie C, les nœuds déclarés (dans la partie A) sont définis. La définition d'un nœud de calcul consiste à donner un identificateur à ce nœud de calcul et à donner les noms des annotations d'entrée, de calcul et de sortie (voir paragraphe 3.2). Ces annotations sont décrites séparément dans la partie B par des **procédures écrites par l'utilisateur**. Cette séparation permet le partage d'un même type d'annotation par plusieurs nœuds de calcul, ce qui est convenable dans les modèles présentant une régularité. Finalement, la précédence est établie dans la partie D par le moyen de méthodes spéciales, définies dans `andes.h`. Chaque partie est détaillée dans les paragraphes qui suivent. Les paragraphes ci-dessous présentent le "mode d'emploi" de la bibliothèque `andes.h`.

### 4.1.1 Déclaration des nœuds de calcul

Un nouveau type est défini dans la bibliothèque `andes.h` pour modéliser les nœuds de calcul : il s'agit du type `Node`. Ce type permet la création de variables qui représentent les nœuds de calcul du modèle. Son utilisation est analogue à l'utilisation des types standard, comme `int` et `float` :

```
...
Node    node_1, node_2, node_3;
Node    grid[MAX_GRID_SIZE][MAX_GRID_SIZE];
Node*   pointer_to_node;
...
```

Comme on l'a vu dans l'exemple ci-dessus, il est possible de construire des tableaux de nœuds de calcul et d'avoir des pointeurs.

### 4.1.2 Définition des nœuds de calcul

Un nœud de calcul est composé d'une annotation d'entrée, d'une annotation de sortie et d'une annotation de calcul. Ces annotations sont définies par le moyen de procédures dans la partie B d'un modèle *ANDES-C*. La fonction de création d'un nœud de calcul s'appelle `comp_node`. Cette fonction est définie dans la bibliothèque `andes.h`. Elle retourne un nœud de calcul. Ses paramètres sont :

1. un identificateur de nœud de calcul (une chaîne de caractères);
2. le nom de la procédure (définie par l'utilisateur dans la partie B d'un modèle *ANDES-C*) qui représente l'annotation d'entrée (`NULL`, si le nœud de calcul est un nœud racine);
3. le nom de la procédure (définie par l'utilisateur dans la partie B d'un modèle *ANDES-C*) qui représente l'annotation de calcul;
4. le nom de la procédure (définie par l'utilisateur dans la partie B d'un modèle *ANDES-C*) qui représente l'annotation de sortie (`NULL`, si le nœud est du type feuille).

Par exemple :

```
...
/* partie A */
Node node_1;
...
/* partie B */
void def-annotation-entree (n)
Node n;
```

```
{ .... }
void def-annotation-calcul (n)
Node n;
{ .... }
void def-annotation-sortie (n)
Node n;
{ .... }

main () {
  ...
  /* partie C */
  node_1 = comp_node ("node#1",
                      def-annotation-entree,
                      def-annotation-calcul,
                      def-annotation-sortie);
  ...
}
```

Pour définir un nœud de calcul, on est obligé d'utiliser `comp_node` (une méthode fournie par `andes.h`). Par contre, les procédures `def-annotation-entree`, `def-annotation-calcul` et `def-annotation-sortie` sont définies par l'utilisateur. Le paramètre `n` (du type `Node`) de ces procédures est obligatoire. Ce paramètre sert de lien entre la fonction `comp_node` et les procédures de définition des annotations. `n` sera utilisé dans ces procédures pour aider la caractérisation des annotations (voir les paragraphes suivants).

### 4.1.3 Définition des annotations d'entrée

Les annotations d'entrée sont définies dans les procédures écrites par l'utilisateur dans la partie B d'un modèle *ANDES-C*. Le type de procédure doit être `void`. La liste de paramètres de ces procédures doit obligatoirement comporter un seul paramètre du type `Node` :

```
void une-autre-def-annotation-entree (n)
Node n;
{ /* corps de la procedure */
  .... }
```

Dans le corps de la procédure de définition d'une annotation d'entrée, il faut définir l'identificateur de l'annotation, son coût et son descripteur de type (voir paragraphe 3.2.2). Ces caractéristiques sont définies par le moyen de la fonction `type_input`, définie dans `andes.h`. `type_input` est obligatoirement présente dans une annotation d'entrée. Cette fonction comporte trois paramètres au minimum :

1. l'identificateur;

2. le nœud de calcul associé (représenté par le paramètre `n` de la procédure `une-autre-def-annotation-entree` dans l'exemple ci-dessus);

3. le descripteur de type.

D'autres paramètres peuvent être ajoutés selon le type d'annotation. Par exemple, une annotation d'entrée du type `AND` avec 10 entrées est modélisée de la façon suivante :

```
...
void une-autre-def-annotation-entree (n)
Node n;
{
   type_input ("annot_entree", n, AND, 10);
}
...
```

Les descripteurs de type disponibles dans `andes.h` sont `SINGLE`, `AND`, `OR`, `DIFF` et `RASS` (respectivement, une annotation d'entrée unique, de types booléens `AND` et `OR` et d'opérations globales de diffusion et de rassemblement). Par exemple,

```
/* Figure (a) : 5 entrees du type OR */
type_input ("annot_entree", n, OR, 5);

/* Figure (b) : 1 entree simple */
type_input ("annot_entree", n, SINGLE);

/* Figure (c)                                  */
/* entree resultat d'une operation globale de */
/* rassemblement de 5 elements d'entree      */
type_input ("annot_entree", n, RASS, 5);

/* Figure (d)                                  */
/* entree resultat d'une operation globale de */
/* diffusion                                   */
type_input ("annot_entree", n, DIFF);
```

Les exemples ci-dessus sont présentés de façon graphique dans la Figure 4.1.

Les descripteurs `SINGLE` et `DIFF` sont associés aux annotations ayant une entrée simple. Les **coûts de communications** associés aux annotations d'entrée chez les nœuds de calcul récepteurs sont définis dans les annotations de sortie chez les nœuds de calcul émetteurs. Ceci a été choisi afin d'éviter la duplication d'informations dans le modèle.

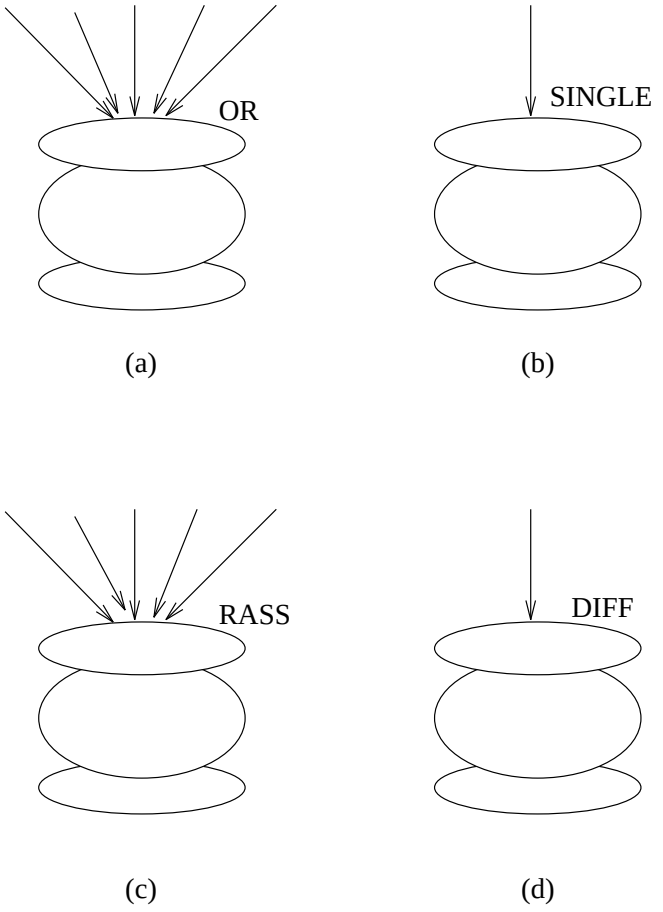



Figure 4.1 : Exemples d'annotation d'entrée.

### 4.1.4 Définition des annotations de sortie

Les annotations de sortie sont analogues aux annotations d'entrée. Il s'agit aussi de procédures définies par l'utilisateur du type `void` avec un paramètre du type `Node`. Dans le corps de la procédure, la fonction `type_output` (définie dans la bibliothèque `andes.h`) est obligatoire. Cette fonction comporte les paramètres suivants :

1. l'identificateur;
2. le nœud de calcul associé (représenté par le paramètre du type `Node`);
3. le descripteur de type;
4. le coût.

Dans l'exemple ci-dessous, on spécifie 10 sorties du type `AND` avec un coût de communication de 2 entiers pour chaque communication :

```
...
void une-autre-def-annotation-sortie (n)
Node n;
{
   type_output ("out", n, AND, 10, int(2));
}
...
```

D'autres types de sortie existent (voir le chapitre 3) : `SINGLE`, `OR-PROB`, `OR-DET`, `OR-IND`, `DIFF` et `RASS`.

```
/* Figure (a)                                    */
/* 3 sorties du type OR probabilise' dependant.  */
/* Une d'entre elles sera choisie : une communi- */
/* cation a 3 entiers ou une communication a 4   */
/* entiers ou une communication a 5 entiers      */
/* 0,5 + 0,2 + 0,3 = 1,0                         */
type_output ("out", n, OR-PROB, 3,
            0.5, int(3), 0.2, int(4), 0.3, int(5));

/* Figure (b)                                    */
/* 3 sorties du type OR probabilise' independant.*/
/* Une communication a 3 entiers et/ou une co-   */
/* mmunication a 4 entiers et/ou une communica-  */
/* tion a 5 entiers                              */
type_output ("out", n, OR-IND, 3,
            0.5, int(3), 0.5, int(4), 0.5, int(5));

/* Figure (c)                                    */
/* 2  sorties du type OR deterministe :          */
/* la premiere depend de 2 entrees (0 et 1) avec */
/* un cout de 3 entiers.                         */
/* La deuxieme depend de 1 entree (2) avec un    */
/* cout de 5 entiers.                            */
type_output ("out", n, OR-DET,
            2,          /* 2 sorties */
            2, 0, 1,    /* 2 entrees : 0 & 1 */
            int(3),     /* cout */
            1, 2,       /* 1 entree : 2 */
            int(5));    /* cout */

/* Figure (d)                                 */
/* 5 sorties correspondantes a la diffusion */
/* d'un vecteur de 10 reels                   */
type_output ("out", n, DIFF, 5, real(10));

/* Figure (e)                              */
/* 1 sortie correspondante a une donnee */
/* a etre rassemblee                       */
```

```
type_output ("out", n, RASS, real(10));
```

Les exemples ci-dessus sont présentés de façon graphique dans la Figure 4.2.

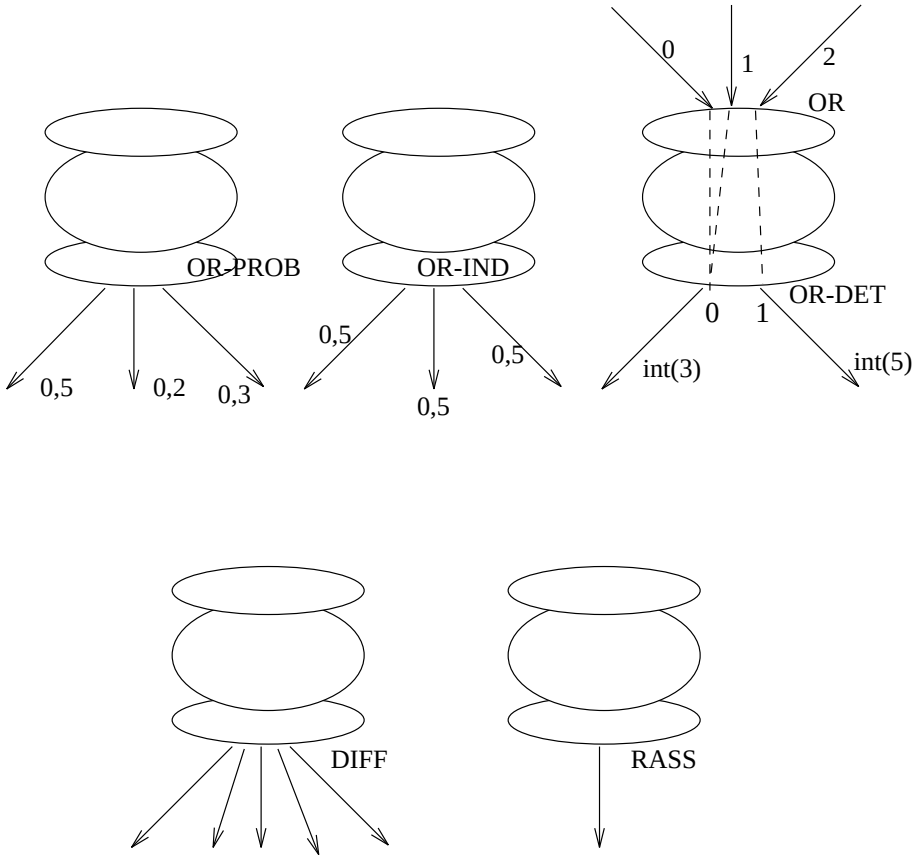



Figure 4.2 : Exemples d'annotation de sortie.

Les fonctions `int` et `real` sont décrites dans la bibliothèque `andes.h`. Elles sont utilisées pour caractériser le type et la quantité de données communiquées entre deux nœuds de calcul. Elles ont un entier positif comme paramètre. Finalement, les coûts et les probabilités liés aux annotations peuvent être des variables dont les valeurs sont connues seulement lors de l'exécution du modèle (voir paragraphe 4.2).

### 4.1.5 Définition des annotations de calcul

La définition d'une annotation de calcul est faite de façon similaire à la définition des annotations d'entrée et de sortie. La définition d'une annotation de calcul est une

procédure écrite par l'utilisateur et avec un paramètre du type `Node`. La fonction employée dans cette procédure est `type_comp` (définie dans `andes.h`). Les paramètres de `type_comp` sont :

1. un identificateur;
2. le nœud de calcul associé (représenté par le paramètre du type `Node`);
3. le coût.

Dans la procédure de définition d'une annotation de calcul, un nouveau *DG-ANDES* peut être spécifié. Cette caractéristique est liée à la possibilité de construire de modèles hiérarchiques (voir le paragraphe 4.3).

L'annotation ci-dessous modélise un coût de calcul de 10000 opérations entières plus 500 opérations du type virgule flottante :

```
...
void une_autre_def_annotation_calcul (n)
Node n;
{
   type_oper ("comp", n, int_ops(10000)+float_ops(500));
}
...
```

Les fonctions `int_ops` et `float_ops` sont décrites dans la bibliothèque `andes.h`. Elles permettent caractériser la quantité et le type des opérations réalisées par un nœud de calcul. Elles ont un entier positif comme paramètre.

### 4.1.6 Définition des précédences

La précédence est définie par le moyen de la fonction `prec`. Cette fonction est définie dans `andes.h` et comporte 5 paramètres :

1. l'identificateur de la précédence;
2. la variable modélisant le nœud de calcul émetteur;
3. le numéro de la porte de sortie chez le nœud de calcul émetteur;
4. la variable modélisant le nœud de calcul récepteur;
5. le numéro de la porte d'entrée chez le nœud de calcul récepteur.

Par exemple, dans :

```
prec ("prec4", diamond[i][j], 0, diamond[i+1][j], 1);
```

la précédence `prec4` est définie entre `diamond[i][j]` et `diamond[i+1][j]`. La communication sort par la porte 0 chez le premier nœud et arrive à la porte 1 chez le deuxième nœud. Les portes représentent une identification individuelle de chaque sortie et de chaque entrée.

## 4.2 Les coûts aléatoires et dépendants

Le nombre d'opérations de calcul et le nombre de valeurs communiquées peuvent être aléatoires et dépendantes. Pour les coûts aléatoires, on utilise les fonctions de variables aléatoires :

```
type_oper ("comp", n, normal (int_ops(10000), int_ops(20)));
```

`normal` est une fonction aléatoire qui génère des numéros selon une distribution normale (le premier paramètre est la moyenne et le deuxième est la variance). Actuellement, il est laissé à l'utilisateur la construction des générateurs de variables aléatoires.

Pour les coûts dépendants, il est nécessaire d'accéder aux attributs d'un nœud de calcul. Ces attributs sont `NOID` (numéro du nœud de calcul), `NOPS` (coût de calcul), `NINP` (nombre d'entrées), `NOUT` (nombre de sorties) et `data_size_out [x]` (coût de communication de la sortie `x`, $0 \leq$ `x` < `NOUT`).

```
type_oper ("comp", n, 10*(node_1->data_size_out[3]));
```

Les attributs sont accessibles par le moyen de l'opérateur `->` d'accès aux structures.

## 4.3 Hiérarchie, récursivité et les structures régulières

Grâce aux procédures disponibles dans le langage C, les descriptions hiérarchiques et récursives peuvent être construites. La récursivité peut être considérée comme un type particulier de hiérarchie où le même modèle se répète à chaque niveau de la structure hiérarchique. Une annotation de calcul peut contenir un *DG-ANDES* (voir paragraphe 3.2.1). C'est avec cette caractéristique que l'on construit des modèles hiérarchiques.

Par exemple, le modèle quantitatif d'un algorithme parallèle peut être décrit comme un seul nœud de calcul :

```
Node niveau_0;
void main () {
...
niveau_0 = comp_node
           ("tache-niveau0", NULL, annot_calc_01, NULL);
...
}
```

`annot_calc_01` spécifie soit un coût de calcul de 10 opérations entières :

```
...
void annot_calc_01 (n)
Node n;
{
   type_oper ("calc-niveau0", n, int_ops (10));
}
```

soit un autre *DG-ANDES* :

```
...
void annot_calc_01 (n)
Node n;
{
   Node inp_node_01, out_node_02,
        node_01_niveau_1, node_02_niveau_1, ....;

   inp_node_01 = input_node
        ("input_node", annot_calc_02, annot_output_01);
   node_01_niveau_1 = comp_node (...);
   node_02_niveau_1 = comp_node (...);
   node_03_niveau_1 = comp_node (...);
   out_node_02 = output_node
        ("output_node", annot_input_01, annot_calc_03);

   prec (...);
   prec (...);
}
```

`input_node` et `output_node` sont deux nouvelles fonctions (définies dans `andes.h`) qui permettent de spécifier les caractéristiques des nœuds de calcul d'entrée et de sortie lors d'un raffinement (voir paragraphe 3.2.3). Ces fonctions ont trois paramètres – toujours un identificateur et :

1. une annotation de calcul et une annotation de sortie pour `input_node` (dans l'exemple ci-dessus, respectivement `annot_calc_02` et `annot_output_01`), ou

2. une annotation d'entrée et une annotation de calcul pour `output_node` (dans l'exemple ci-dessus, respectivement `annot_input_01` et `annot_calc_03`).

`input_node` et `output_node` sont obligatoires dans un nœud de calcul raffiné. Ces fonctions ont un nombre différent de paramètres relativement à `comp_node` (défini comme dans le paragraphe 3.2.3). Par exemple, supposons que le nœud de calcul $X$ est raffiné par un *DG-ANDES* dont les nœuds interface d'entrée et de sortie sont respectivement $X\_in$ et $X\_out$ :

- l'annotation d'entrée de $X\_in$ est la même que l'annotation d'entrée de $X$;

- l'annotation de sortie de $X\_out$ est la même que l'annotation de sortie de $X$.

Les structures régulières sont construites en utilisant de simples procédures avec paramètres. Par exemple, la procédure pour la construction d'un arbre binaire équilibré de nœuds de calcul avec un paramètre $p$ (indiquant la profondeur de l'arbre) peut être :

```
void arbre_binaire (entree, sortie, p)
Node  entree,    % noeud racine de l'arbre
                 % deja defini hors la procedure
      sortie;    % noeud de fin de l'arbre
                 % deja defini hors la procedure
int    p;
Node** noeuds;

{
   % construction de l'arbre
}
```

L'implémentation de ces structures régulières est cachée à l'utilisateur.

## 4.4 Un exemple simple

Supposons le *DG-ANDES* de la Figure 4.3. Ce graphe est composé de quatre nœuds de calcul : `chef1`, `ouvrier[0]`, `ouvrier[1]`, `ouvrier[2]` et `chef2`. `chef1` et `chef2` ont chacun un coût de calcul de 10 opérations entières. Les nœuds `ouvrier` ont chacun un coût de calcul de 100 opérations entières. 144 entiers sont échangés entre les nœuds `chef` et chacun des nœuds `ouvrier`.

D'abord, il faut déclarer les nœuds de calcul :

```
#include <andes.h>

Node chef1, chef2, ouvrier[3];
```

Ensuite, il faut définir les annotations d'entrée, de sortie et de calcul. Il y a deux types d'annotation d'entrée : `SINGLE` (les entrées des ouvriers) et `AND` (l'entrée de `chef2`).

```
void entree_ouvrier (n)
Node n;
{
   type_input ("entree_ouvrier", n, SINGLE);
}

void entree_chef2 (n)
Node n;
{
   type_input ("entree_chef2", n, AND, 3);
}
```

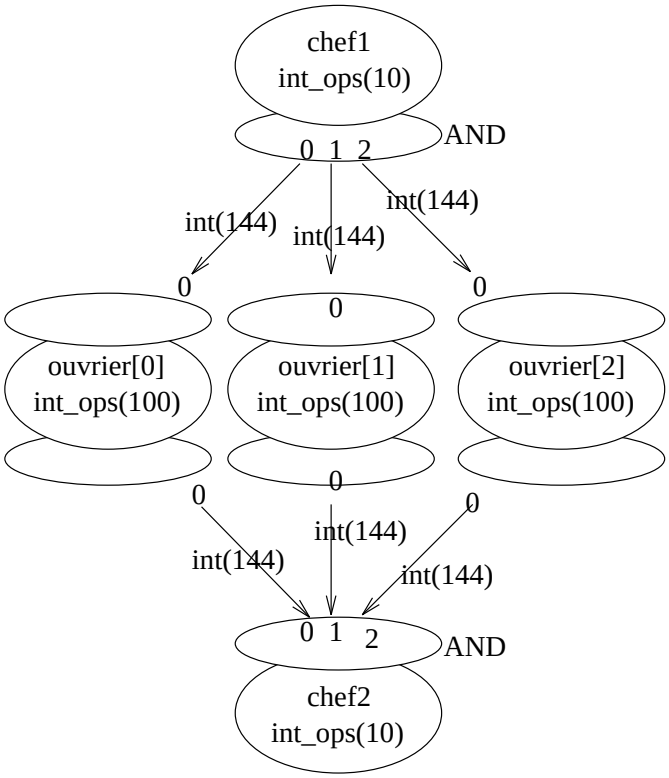



Figure 4.3 : Exemple simple de l'utilisation de la bibliothèque `andes.h`.

Il y a deux types d'annotation de sortie : `AND` (la sortie de `chef1`) et `SINGLE` (la sortie de chaque ouvrier).

```
void sortie_chef1 (n)
Node n;
{
   type_output ("sortie_chef1", n, AND, 3, int(144));
}

void sortie_ouvrier (n)
Node n;
{
   type_output ("sortie_ouvrier", n, SINGLE, int(144));
}
```

Finalement, il y a deux annotations de calcul à spécifier :

```
void calcul_chef (n)
Node n;
{
```

```
   type_oper ("calcul_chef", n, int_ops (10));
}

void calcul_ouvrier (n)
{
   type_oper ("calcul_ouvrier", n, int_ops (100));
}
```

La description continue avec la définition des nœuds de calcul dans la procédure `main` :

```
int main () {

   int ind;

   chef1 = comp_node ("chef_1", NULL, calcul_chef, sortie_chef1);
   chef2 = comp_node ("chef_2", entree_chef2, calcul_chef, NULL);

   for (ind=0; ind<3; ind++)
      ouvrier[ind] = comp_node ("ouvrier[ind]",
                                entree_ouvrier,
                                calcul_ouvrier,
                                sortie_ouvrier);
```

Finalement, la précédence entre les nœuds de calcul est définie :

```
   for (ind=0; ind<3; ind++) {
      prec ("prec_chef_ouvrier", chef1, ind, ouvrier[ind], 0);
      prec ("prec_ouvrier_chef", ouvrier[ind], 0, chef2, ind);
   }
}
```

Cet exemple simple aidera la compréhension des exemples plus complexes présentés dans les paragraphes suivants.

## 4.5 *ANDES*, les paradigmes de calcul et exemples

Les paragraphes précédents présentent le mode d'emploi de la bibliothèque `andes.h` pour la représentation d'un modèle *ANDES*. Tous les modèles *ANDES* sont des graphes orientés sans circuit, où chaque sommet est composé d'annotations d'entrée, de sortie et de calcul. Ainsi, notre vision est orientée vers la représentation du parallélisme de contrôle. Plusieurs modèles de calcul et de programmation peuvent être modélisés en utilisant cette technique. En général, l'utilisateur peut donner aux annotations d'entrée et de sortie une sémantique particulière au paradigme de calcul employé. Par exemple, dans un modèle basé sur l'échange de messages, on peut avoir : `RECV-ALL` (traduit par `AND`), si un nœud de calcul doit recevoir des messages de tous les nœuds de

calcul prédécesseurs; `ALT` (traduit par `OR`), si un nœud de calcul démarre dès qu'un message arrive, etc…. Un exemple de représentation de programmes faisant des appels de procédures à distance est présenté dans le paragraphe 4.5.4. Dans cet exemple, on utilise des équivalents (via `define`) pour modéliser les types de base caractéristiques du paradigme basé sur les appels de procédures à distance.

Nous présentons quelques exemples de modélisation d'algorithmes. D'abord nous présentons le modèle de l'algorithme de résolution d'un système linéaire triangulaire, utilisant la méthode de substitution (en anglais *back substitution*) [BT89]. Ensuite, nous analysons le modèle de l'algorithme de Strassen pour la multiplication rapide de matrices [CT93]. Nous utilisons la récursivité pour construire le modèle de l'algorithme de Strassen. Un exemple d'utilisation d'annotations d'entrée, de sortie et de calcul est donné dans un exemple issu de la programmation logique. Pour montrer que *ANDES* peut être utilisé dans la modélisation de programmes qui suivent un modèle de programmation autre que l'échange de messages, une description d'un programme parallèle utilisant des appels de procédures à distance (en anglais *RPC - Remote Procedure Call*) est présentée.

### 4.5.1 La *substitution en arrière*

Cet algorithme (en anglais *back substitution*) correspond à une méthode de résolution de systèmes linéaires triangulaires. La structure des nœuds de calcul dans le modèle est présentée dans la Figure 4.4.

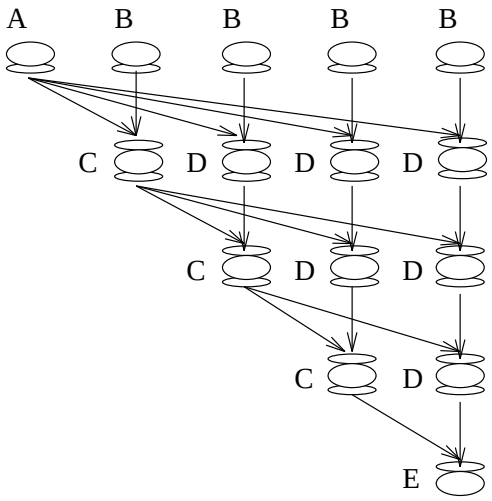



Figure 4.4 : La structure du modèle *ANDES* pour l'algorithme de substitution.

Un extrait du modèle *ANDES-C* est présenté ci-dessous. Le nombre d'éléments du système est choisi uniformément entre `MIN_TASKS` et `MAX_TASKS` (entre 100 et 200

dans l'exemple) :

```
#include <andes.h>

#define MIN_TASKS 100
#define MAX_TASKS 200

...

void main() {

...
   /* choix du nombre de taches */
   srand48 (); /* initialisation du generateur de nombres
                  pseudo-aleatoires */
   /* lrand48 - generateur de nombres pseudo-aleatoires entre 0
      et l'infini */
   numb_tasks = MIN_TASKS + (lrand48()%(MAX_TASKS-MIN_TASKS));

   /* construction du tableau des noeuds de calcul */
   /* tableau dynamique de numb_tasks versus numb_tasks elements */
   task_graph = (Node **) malloc
                (numb_tasks * sizeof (Node *));
   for (i=0; i<numb_tasks; i++)
      task_graph[i] = (Node *) malloc
                ((numb_tasks-i) * sizeof (Node));

...
```

Les nœuds du modèle sont représentés par la variable `task_graph`. Les nœuds sont ainsi définis :

```
   for (i=0; i<numb_tasks; i++) {
   /* pour chaque etage i du graphe */

      if (i == 0)
         for (j=0; j<numb_tasks; j++)
            if (j == 0) {          /* noeud A de la figure */
               sortie = numb_tasks-1;
               task_graph[i][j] = comp_node
                         ("A", NULL, comp, totasks);
            }
            else {                 /* noeuds B de la figure */
               sortie = 1;
               task_graph[i][j] = comp_node
                         ("B", NULL, comp, totasks);
            }
      else
         if (i == (numb_tasks-1)) {
            entree = 2;            /* noeud E de la figure */
            task_graph[i][0] = comp_node
```

```
                        ("E", fromtasks, comp, NULL);
         }
         else
            for (j=0; j<(numb_tasks-i); j++)
               if (j == 0) {       /* noeud C de la figure */
                  entree = 2;
                  sortie = (numb_tasks-i)-1;
                  task_graph[i][j] = comp_node
                         ("C", fromtasks, comp, totasks);
               }
               else {              /* noeuds D de la figure */
                  entree = 2;
                  sortie = 1;
                  task_graph[i][j] = comp_node
                         ("D", fromtasks, comp, totasks);
               }
   }
```

La précédence est simple à représenter. La première tâche d'un étage communique avec toutes les autres tâches du même étage :

```
      if (i != 0)
         for (j=0; j<(numb_tasks-i); j++) {
            prec ("diffusion", task_graph[i-1][0], j,
                                task_graph[i][j], 0);
            prec ("suite", task_graph[i-1][j+1], 0,
                           task_graph[i][j], 1);
         }
```

L'exemple ci-dessus donne une idée de comment on construit le corps d'une description textuelle *ANDES*. Les détails concernant l'algorithme de substitution peuvent être trouvés dans [BT89].

### 4.5.2 L'algorithme de Strassen

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 Figure 4.5.

L'aspect important à remarquer est la **récursivité** du modèle : une procédure `strassen` est appelée pour chaque niveau `n` de la récursion jusqu'à ce qu'un seuil soit achevé. Au-delà de ce seuil, les multiplications de sous-matrices sont faites séquentiellement. Voici les définitions des nœuds de calcul :

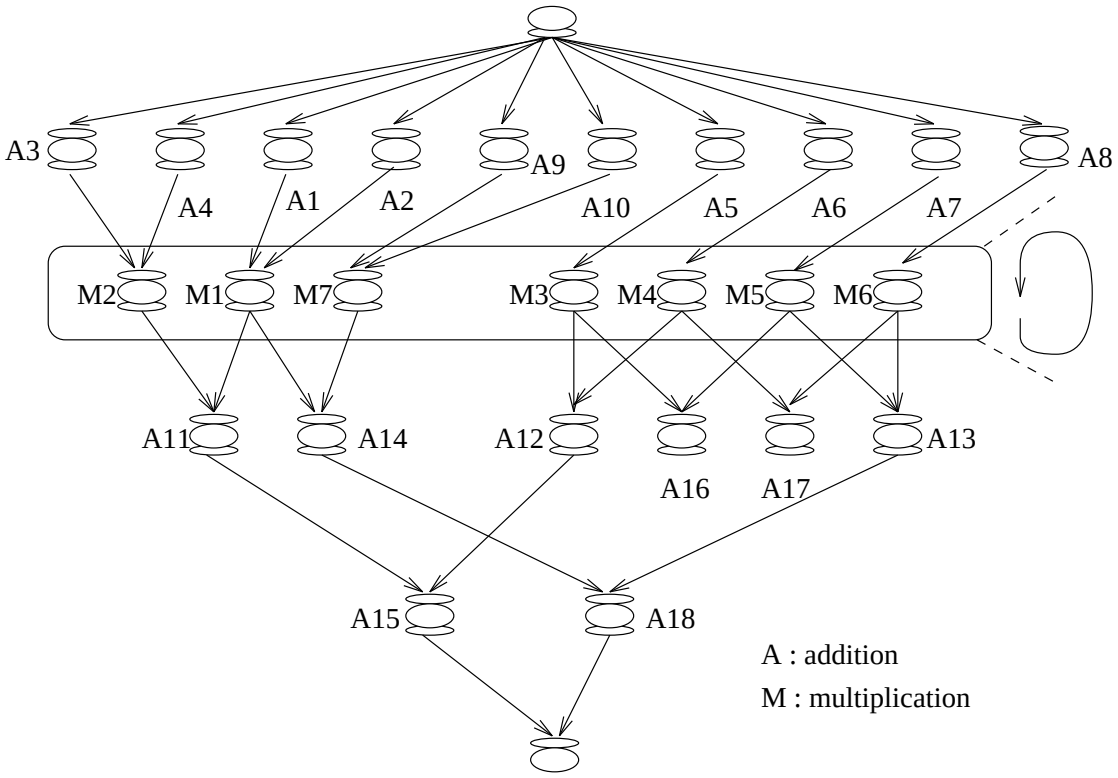



Figure 4.5 : Structure des nœuds de calcul de l'algorithme de Strassen.

```
...
void strassen (n, begin, finish)
int  n;             /* niveau de l'hierarchie */
Node begin, finish; /* debut et fin de la multiplication */
{

   /* les elements des matrices sont des entiers */

   Node             addition[18]; /* taches d'addition */
   int              i;

   /* definition de A1, A2, ..., A10 */
   for (i=0; i<10; i++) {
      entree = 1;
      sortie = 1;
      comput = int ((n * n)/4);
      addition[i] = comp_node
            ("addition-1", fromtasks, comp, totasks);
   }
```

```
   /* definition de A11, 12, ..., A18 */
   for (i=10; i<18; i++) {
      entree = 2;
      sortie = 1;
      comput = int ((n * n)/4);
      if ((i != 11) && (i != 12))
         addition[i] = comp_node
               ("addition-2", fromtasks, comp, totasks);
      else {
         data_size_out1 = int ((n * n)/4);
         addition[i] = comp_node
              ("addition-3", fromtasks, comp, totask_comm);
      }
   }
...
```

Si le seuil n'est pas encore achevé, la procédure `strassen` est appelée de nouveau :

```
...
      for (i=0; i<7; i++)
         strassen (n/2, start[i], end[i]);
...
```

Si le seuil est achevé, les nœuds du niveau plus bas de l'hiérarchie sont décrits (une communication du type `CONTEXT` indique que les nœuds de calcul émetteur et récepteur doivent être placés sur le même processeur - cf. le chapitre 6) :

```
...
   else {
      Node multiplication[7];  /* taches de multiplication */

      /* definition de M1 */
      entree = 2;
      sortie = 2;
      comput = int ((n * n * n)/8);
      data_size_out1 = int ((n * n)/4);
      data_size_out2 = int ((n * n)/4);
      multiplication[0] = comp_node
              ("mult-1", fromtasks, comp, totask_comm);

      /* definition de M2  et M7 */
      entree = 2;
      sortie = 1;
      comput = int ((n * n * n)/8);
      multiplication[1] = comp_node
              ("mult-2", fromtasks, comp, totasks);
      multiplication[6] = comp_node
              ("mult-7", fromtasks, comp, totasks);

      for (i=2; i<6; i++) {
```

```
         entree = 1;
         sortie = 2;
         comput = int ((n * n * n)/8);
         if (i == 2)
            /* definition de M3 */
            multiplication[i]   = comp_node
                        ("mult-3", fromtasks, comp, totasks);
         else
            if (i == 3) {
               data_size_out1 = int ((n * n)/4);
               data_size_out2 = int ((n * n)/4);
               /* definition de M4 */
               multiplication[i] =
                   comp_node
                     ("mult-4", fromtasks, comp, totask_comm);
            }
            else
               if (i == 4) {
                  data_size_out1 = (n * n)/4;
                  data_size_out2 = CONTEXT;
                  /* definition de M5 */
                  multiplication[i] = comp_node
                     ("mult-5", fromtasks, comp, totask_comm);
               }
               else {
                  data_size_out1 = CONTEXT;
                  data_size_out2 = (n * n)/4;
                  /* definition de M6 */
                  multiplication[i] = comp_node
                     ("mult-6", fromtasks, comp, totask_comm);
               }
      }
...
```

Le corps principal de la description est donné ci-dessous :

```
/*************************************************************/
/* Description                                               */
/*************************************************************/

void main () {

   if ((MATRIX_SIZE > THRESHOLD) && ((MATRIX_SIZE%2) == 0)) {

      Node start, end;
      ...
      start = comp_node ("START", NULL, comp, totask_comm);
      ...
      end   = comp_node ("END", fromtasks, comp, NULL);
```

```
      /* DECOMPOSITION RECURSIVE */
      strassen (MATRIX_SIZE, start, end);
   }
   else {
      /* cas degenere : il y a un seul noeud de calcul
         dans le modele */
      Node only_one;
      comput = (MATRIX_SIZE*MATRIX_SIZE*MATRIX_SIZE)/8;
      only_one = comp_node ("BIG_ONE", NULL, comp, NULL);
   }
}
```

Dans l'exemple ci-dessus, la décomposition récursive a été représentée par des appels récursifs de procédures.

### 4.5.3 Prolog

Cet exemple et le prochain (le modèle d'un algorithme de division pour paralléliser développé selon un modèle de calcul basé sur les appels de procédures à distance) sont utilisés pour mieux comprendre la définition des annotations d'entrée, de sortie et de calcul.

Le paradigme de programmation logique [ABZ92] est basé sur les faits (e.g., "Marie est la mère de Pierre") et un ensemble de règles qui produisent d'autres faits (e.g., "Si $X$ est père, alors $X$ est un homme"). Le flot de contrôle d'un programme logique est déterminé par un système qui répond à des questions concernant les faits et les règles (e.g., "Qui est la mère de Brigitte?" ou "Est-ce que Jean est le père de Marie?"). Ce système (la machine d'inférence) utilise plusieurs stratégies pour déterminer la réponse aux questions. En général, la précédence des activités de la machine d'inférence peut être modélisée par un arbre (Figure 4.6). Chaque nœud de calcul peut démarrer en parallèle plusieurs flots de contrôle, un pour chaque règle à tester. Un flot de contrôle finit lorsque la question est répondue ou lors de l'occurrence d'un échec. Le nombre de fils de chaque nœud de calcul est déterminé par les faits et par les règles du programme logique.

Les annotations d'entrée sont soit du type `AND` (pour le nœud de calcul racine du graphe) soit du type `SINGLE`. `fin` est le nombre de feuilles de l'arbre généré) :

```
void fromtasks (n)
Node n;
{
   type_input ("inp-single", n, SINGLE);
}

void fromtasksn (n)
Node n;
```

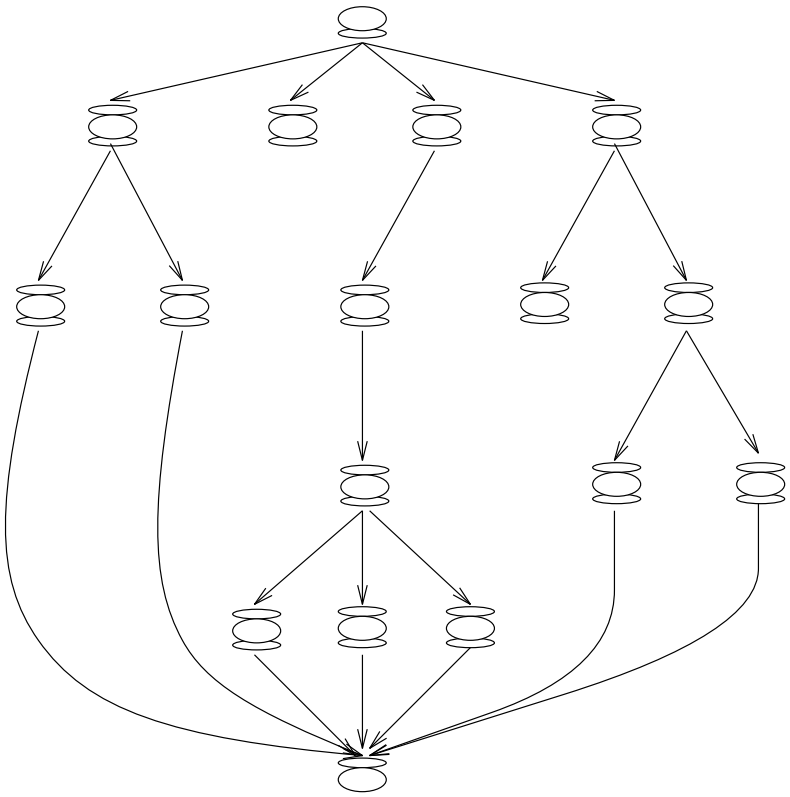


Figure 4.6 : Structure de la résolution d'un programme logique.

```
{
   type_input ("inp-and", n, AND, fin);
}
```

Il y a deux types d'annotations de sortie : une est du type `AND` (correspondant aux fils d'un nœud de calcul) et l'autre du type `SINGLE` (une sortie seulement - lorsque le flot de contrôle finit). Dans le modèle textuel, un seul type d'annotation de sortie est instancié plusieurs fois par le moyen de la variable globale `number_sons` :

```
int number_sons;

void totasks (n)
Node n;
{
   type_output ("out-and", n, AND, number_sons, int(1));
}
```

```
void totasks1 (n)
Node n;
{
   type_output ("out-single", n, SINGLE, int(1));
}
```

La variable `number_sons` est nécessaire, car on ne connaît pas d'avance le nombre de fils d'un nœud de calcul. Ce nombre est déterminé lors de l'exécution du modèle.

Enfin, l'annotation de calcul est représentée normalement par un coût correspondant à la comparaison d'un fait avec une règle. Dans l'exemple ci-dessous, le coût correspond à 100 opérations entières.

```
void comp (n)
Node n;
{
   type_oper ("comp", n, int(100));
}
```

Naturellement, des coûts dépendants ou probabilisés pourraient être utilisés.

### 4.5.4 Diviser pour paralléliser

L'utilisateur peut créer de nouveaux types d'annotations d'entrée et de sortie selon le modèle de calcul suivi par les algorithmes représentés dans *ANDES*. Par exemple, la Figure 4.7 montre un modèle de programme de division pour paralléliser en utilisant des annotations avec des types particuliers à Athapascan [Chr94], langage de programmation du projet APACHE (voir le chapitre 1) basé sur un modèle d'appels de procédures à distance. Un algorithme du type "division pour paralléliser" décompose un grand problème en sous-problèmes qui sont résolus indépendamment. Un sous-problème peut être lui-même décomposé.

Les nouveaux types d'annotation sont `SPAWN` qui modélise l'appel à une procédure et `RES` qui correspond à rendre un résultat vers la procédure appelante. L'annotation `WAIT` modélise une attente pour un résultat et l'annotation `REQ` modélise l'arrivée d'une requête. Comme dans l'exemple précédent concernant les programmes logiques, les annotations d'entrée, de sortie et de calcul sont instanciées à travers des variables globales (dans cet exemple, `sortie`, `entree` et `comput`). La sémantique des paramètres de `SPAWN` est analogue à la sémantique de l'annotation booléenne `ANDR` :

```
/*************************************************************/
/* Variables globales                                        */
/*************************************************************/

#define RES   SINGLE
#define SPAWN AND
#define WAIT  AND
```

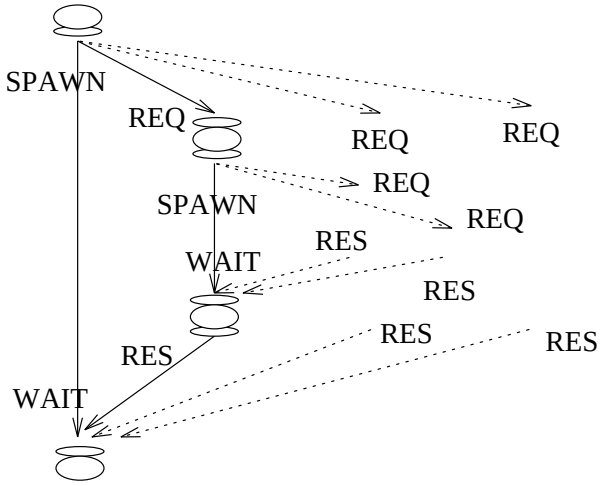



Figure 4.7 : Division pour paralléliser - modèle graphique.

```
#define REQ   SINGLE

int sortie,
    entree,
    comput;

/*************************************************************/
/* Les annotations de sortie                                 */
/*************************************************************/

void totasks (n)
Node n;
{
   if (sortie == 1)
      type_output ("result", n, RES, 4);
   else
      type_output ("spawn",  n, SPAWN, sortie,
                                       sortie-1, 4,
                                       1,        SUITE);
}

/*************************************************************/
/* Les annotations d'entree                                  */
/*************************************************************/

void wait (n)
Node n;
{
   type_input ("wait", n, WAIT, entree);
}
```

```
void req (n)
Node n;
{
   type_input ("req", n, REQ, entree);
}

/*************************************************************/
/* Les annotations de calcul                                 */
/*************************************************************/

void comp (n)
Node n;
{
   type_oper ("comp", n, comput);
}

...
```

La construction du modèle est récursive. Il y a une procédure (`division`) qui est appelée à chaque fois qu'une décomposition doit être faite :

```
/*************************************************************/
/* Procedure "division"                                      */
/* Construction recursive du graphe                          */
/*************************************************************/

void division (deep, begin, finish)
int  deep;
Node begin, finish;
{
   if (deep < DEEP) {
/*************************************************************/
/* Continue ‘a diviser                                       */
/*************************************************************/

      ...
      /* definition des noeuds */
      for (i=0; i<SONS; i++)
         start[i] = comp_node ("ep_01", req, comp, totasks);
      ...
      for (i=0; i<SONS; i++)
         end[i]  = comp_node ("ep_01", wait, comp, totasks);

      for (i=0; i<SONS; i++)
         division (deep+1, start[i], end[i]);

      /* precedence dans ce niveau */
      for (i=0; i<SONS; i++) {
         prec ("prec", begin, i, start[i], 0);
         prec ("prec", end[i], 0, finish, i);
```

```
      }
      prec ("prec", begin, SONS, finish, SONS);
   }
   else {
/*************************************************************/
/* Arrete de diviser                                         */
/*************************************************************/

      ...
      /* definition des noeuds de l'hierarchie plus basse */
      for (i=0; i<SONS; i++)
         sons[i] = comp_node ("ep_02", req, comp, totasks);

      /* precedence dans ce niveau */
      for (i=0; i<SONS; i++) {
         prec ("prec", begin, i, sons[i], 0);
         prec ("prec", sons[i], 0, finish, i);
      }
      prec ("prec", begin, SONS, finish, SONS);
   }
}
```

Le corps principal du modèle consiste à définir les nœuds initial et final et appeler la procédure de division :

```
void main () {

   ...
   start  = comp_node ("ep_01", NULL, comp, totasks);
   ...
   end    = comp_node ("ep_01", wait, comp, NULL);
   division (deep+1, start, end);
}
```

Il faut remarquer que la récursivité (et l'hiérarchie) est utilisée lorsque la décomposition d'un nœud de calcul garde les précédences entre ce nœud décomposé et ses nœuds prédécesseurs et successeurs. Dans ce cas, la récursivité est modélisée par la récursivité du langage C.

## 4.6 Conclusions

Ce chapitre a présenté une implémentation de la méthode *ANDES*. Trois alternatives auraient été envisagées pour créer des modèles textuels *ANDES* :

- utiliser un langage spécifique pour la modélisation de graphes;
- développer un nouveau langage;

- utiliser un langage hôte.

Nous avons choisi la troisième option en utilisant C. Ce langage de programmation contient des structures qui sont très adéquates à la modélisation des aspects de *ANDES*, principalement les instructions itératives et les procédures. Le langage C est aussi très connu et il nous a permis de développer de façon rapide un prototype de générateur de charges synthétiques. Dans l'avenir, nous envisageons d'utiliser un langage orienté objets (e.g., C++), puisque les nœuds de calcul peuvent être modélisés comme des objets sur lesquels on peut réaliser certaines opérations.

La bibliothèque `andes.h` est composée de plusieurs fonctions de manipulation de nœuds de calcul (par exemple, `comp_node`, `prec`, `type_oper`). Dans ce chapitre, nous avons présenté la sémantique de l'interface de ces fonctions. Ces fonctions sont des boîtes noires qui sont définies en fonction du type d'utilisation des modèles *ANDES-C*. Par exemple, une bibliothèque `andes.h` peut être construite pour générer la charge de travail donnée à un simulateur ou à un modèle analytique. Dans la suite, nous présentons l'utilisation de *ANDES-C* pour la génération de programmes synthétiques.

# Chapitre 5

# L'Outil *ANDES-Synth*

Ce chapitre a pour but de répondre à la question : "dans quelle mesure le modèle *ANDES* est utile dans un contexte d'évaluation des performances d'algorithmes parallèles?" D'abord, nous présentons un tour d'horizon des techniques classiques disponibles pour l'évaluation des performances. Les avantages et les inconvénients de chaque approche sont ensuite discutés. Le besoin de résultats proches de l'implémentation des programmes parallèles nous a conduit vers une technique basée sur les mesures, mais suffisamment flexible pour permettre l'évaluation rapide d'un grand éventail de types d'algorithmes. L'approche choisie est basée sur les **programmes synthétiques**. Cette approche consiste en une utilisation réelle des ressources, sans la résolution d'un vrai problème de calcul. Néanmoins, comme on l'a vu dans l'introduction du chapitre 4, *ANDES* peut être utilisé aussi avec des outils basés sur d'autres techniques d'évaluation. Dans la bibliothèque `andes.h` de *ANDES-C*, cette flexibilité correspond à différentes définitions de chaque fonction disponible dans cette bibliothèque : chaque fonction est définie selon les besoins des différentes techniques (e.g., modélisation analytique ou simulation). Dans le chapitre 4, nous avons présenté une interface pour la description de modèles *ANDES*. Dans ce chapitre, nous présentons le comportement de chaque fonction vis-à-vis d'une génération de programmes synthétiques. En d'autres termes, nous expliquons à quoi correspond l'exécution d'un modèle exprimé en *ANDES-C*.

Un outil a été construit pour *ANDES* et pour les programmes synthétiques sur la machine cible Méganode (pour plus de détails sur cette machine, voir l'appendice A). Le principe et la structure de cet outil, *ANDES-Synth*, sont ensuite présentés. A titre de comparaison, nous présentons *OLGA*, un autre outil basé sur l'exécution de programmes synthétiques. La conclusion de ce chapitre servira de point de départ au chapitre suivant : l'utilisation de *ANDES-Synth* dans l'évaluation des stratégies de placement statique.

## 5.1 Techniques d'évaluation de performances

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 complexe est la modélisation et plus précis sont les indices obtenus.

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. 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 utilisation de la 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 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 dure 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 tous 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 ci-dessus 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'un jeu de test (en anglais *benchmark*) de la machine cible. `SCANF_TIME` et `PRINTF_TIME` comportent 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* (décrit avec la bibliothèque `andes.h`) et à partir d'un modèle de machine. 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 machine : 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 machine, la génération rapide de programmes synthétiques et la possibilité de tester plusieurs stratégies d'implémentation représentent le cœur de l'outil *ANDES-Synth*.

## 5.2 Le principe de *ANDES-Synth*

L'environnement *ANDES-Synth* est un outil développé au sein du projet *ALPES* (*AL*gorithmes *P*aralléles et *E*valuation de *S*ystèmes). 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 5.1. 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 machine).

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

La structure présentée dans la Figure 5.1 est le pivot de l'outil *ANDES-Synth*. Analysons cette structure en détail.

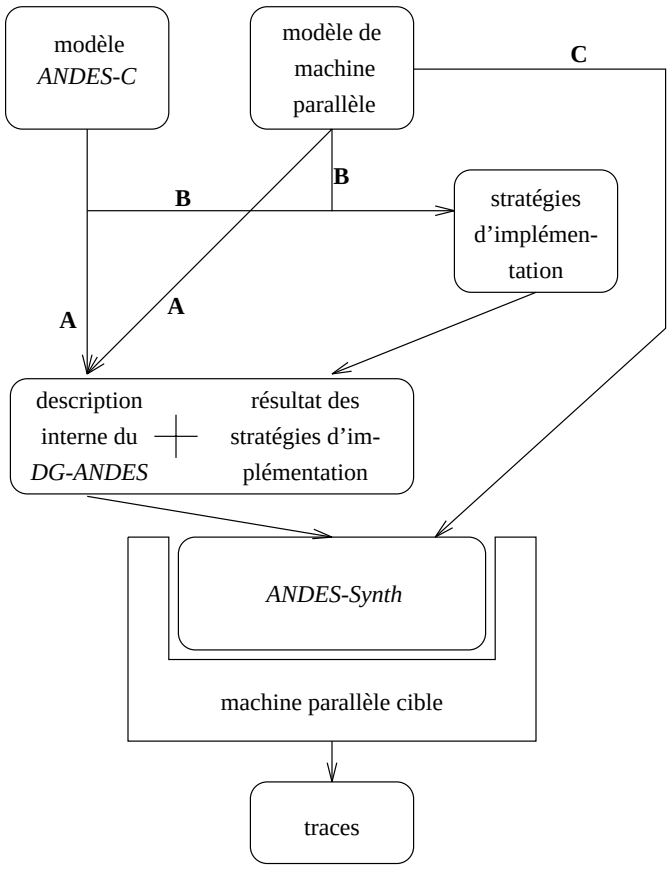



Figure 5.1 : L'environnement *ANDES-Synth*.

### 5.2.1 Le modèle *ANDES-C* et le modèle de machine

Le modèle *ANDES-C* d'un algorithme parallèle est un programme C qui utilise la bibliothèque `andes.h`. La définition des fonctions disponibles dans cette bibliothèque varie selon la technique d'évaluation de performances employée. Dans *ANDES-Synth*, la technique choisie est basée sur l'exécution de programmes synthétiques. De cette façon, l'exécution des fonctions disponibles dans `andes.h` conduit à la génération des informations nécessaires à l'exécution de programmes synthétiques. Ces informations sont les coûts de calcul et de communication (définis par les fonctions `type_oper`, `type_input` et `type_output` de `andes.h`) et la précédence entre les nœuds de calcul (définie par la fonction `prec` de `andes.h`).

Une description dans un format interne de la charge synthétique résulte de la compilation et de l'exécution du programme C correspondant au modèle *ANDES-C*. Le modèle de la machine parallèle intervient dans cette étape. Les coûts de calcul et de communication peuvent être modifiés afin de représenter des processeurs et des réseaux

de communication plus lents ou plus rapides par rapport aux processeurs et au réseau de communication de la machine parallèle cible (voir Figure 5.1). La modification des coûts de calcul afin de modéliser un processeur avec une puissance de calcul différente est directe : par exemple, un coût de calcul réduit à la moitié représente un processeur deux fois plus puissant que le processeur de la machine parallèle cible. Par contre, la transformation du coût de communication afin de représenter un réseau avec une bande passante plus ou moins importante est plus difficile. Plusieurs paramètres du réseau sont considérés : la bande passante des liens, le type de routage et la topologie du réseau. La charge imposée au réseau dépend de ces paramètres. Cette charge doit être prise en compte lors du choix d'un coût de communication pour représenter au mieux le réseau de communication émulé par la machine parallèle cible. Dans notre modèle de machine, les différents réseaux de communication sont représentés par des expressions analytiques qui expriment le temps de communication en fonction des paramètres du réseau [TP94].

Les coûts de calcul et de communication ne sont pas seulement modifiés. Ces coûts sont aussi **traduits** en un certain nombre d'itérations vides (le calcul dans un programme synthétique) et de paquets d'octets communiqués entre les nœuds de calcul (la communication dans un programme synthétique). Si on revient au modèle *ANDES-C*, les coûts de calcul et de communication sont exprimés par des fonctions, par exemple, du type `int_ops` ou `int`. C'est le modèle de machine qui définit combien d'itérations (exécutées sur la machine parallèle cible) représente, par exemple, une opération entière. De même, c'est le modèle de machine qui définit combien d'octets représente, par exemple, une valeur entière à échanger entre deux nœuds de calcul.

Dans la Figure 5.1, l'interaction entre le modèle *ANDES-C* et le modèle de machine est représentée par les arcs **A**.

### 5.2.2 Les stratégies d'implémentation

Notre modèle de programme parallèle est composé de nœuds de calcul qui doivent être exécutés sur une machine composée de plusieurs processeurs. Plusieurs stratégies d'implémentation sont disponibles afin d'obtenir une exécution performante. Ces stratégies peuvent être utilisées avant l'exécution (utilisation statique) ou pendant l'exécution (utilisation dynamique). Les stratégies d'ordonnancement [Ull75], de groupement [YG92] et de placement [Bok81] sont statiques. Le "placement dynamique" (c'est à-dire, le choix pendant l'exécution du processeur) correspond au problème de l'équilibrage de charge (en anglais *load balancing*) [CP92].

Les décisions prises par les stratégies d'implémentation dépendent naturellement du modèle de l'algorithme parallèle et du modèle de la machine parallèle. Chaque stratégie d'implémentation a besoin d'un nombre différent d'informations de chaque modèle. Par exemple, pour les algorithmes classiques d'ordonnancement, il faut connaître du modèle *ANDES-C* :

- le nombre de nœuds de calcul;

- la précédence entre les nœuds de calcul;
- les coûts de calcul;
- les coûts de communication.

Du modèle de machine, il faut connaître :

- le nombre de processeurs;
- la puissance de calcul du processeur;
- les caractéristiques (topologie, bande passante, etc…) du réseau de communication.

Eventuellement, différentes stratégies sont appliquées à partir d'un seul modèle de programme. Par exemple, l'utilisation d'une stratégie de placement implique un modèle de programme basé sur un graphe non orienté. Un modèle *ANDES-C* décrit un graphe orienté. Une solution pour enlever cette incompatibilité de modèles est de grouper le *DG-ANDES* et ensuite placer les groupes générés. Cette démarche est détaillée dans le chapitre 6.

Dans la Figure 5.1, l'interaction entre le modèle *ANDES-C*, le modèle de machine et les stratégies d'implémentation est représentée par les arcs **B**.

### 5.2.3 L'exécution de la charge synthétique

Les étapes d'interaction du modèle *ANDES-C* avec le modèle de machine et l'utilisation des stratégies statiques d'implémentation correspondent à la construction d'une charge synthétique. Comme on l'a vu antérieurement, la différence de ce type de charge par rapport à une charge réelle est que les deux représentent une utilisation effective des ressources de la machine parallèle émulée par la machine cible, mais que la charge synthétique ne résout pas un problème. D'autre part, les ressources sont utilisées de manières différentes : dans l'exécution d'une charge synthétique, les calculs sont faits par des boucles vides. Les contenus des messages échangés n'ont aucune importance : c'est la taille du message (en octets) qui compte.

Il y deux approches possibles pour exécuter la charge synthétique. La première méthode consiste à générer un **vrai programme synthétique**. Par exemple, si la machine cible est basée sur des Transputers, chaque nœud de calcul peut être, par exemple, traduit en un programme Occam2. Au début de chaque processus, la réception de messages est faite. Ensuite la boucle vide est effectuée et finalement les envois vers les nœuds de calcul successeurs ont lieu. C'est le système d'exploitation de la machine cible qui est chargé du routage de messages et de la gestion de la multiprogrammation. Ce système d'exploitation peut être modifié (si possible) selon les paramètres du modèle de machine. Par exemple, le routage de la machine cible est modifié afin

d'émuler le routage de la machine modélisée. Cette implication du modèle de machine est représentée par l'arc **C** dans la Figure 5.1.

Une deuxième approche est le développement d'un **noyau exécutif** qui contrôle l'exécution de la charge synthétique. Ce noyau s'exécute sur le système d'exploitation de la machine cible. Dans cette technique, on ne parle pas de programme synthétique, mais plutôt d'un tableau d'informations de charge qui dirige le comportement du noyau exécutif. Ce noyau exécute les boucles vides correspondantes aux nœuds de calcul au fur et à mesure que les messages arrivent sur un processeur.

Dans la première approche, les programmes synthétiques s'exécutent directement sur le système d'exploitation de la machine cible. Si les coûts de calcul et de communication sont bien choisis, cette exécution doit être similaire (concernant les temps et l'utilisation des processeurs et du réseau de communication) à l'exécution du programme réel, modélisé par le programme synthétique. D'autre part, le noyau exécutif pour la deuxième approche représente une surcharge qui doit être prise en compte. Un autre avantage de la première approche est que lorsqu'une implémentation performante du programme synthétique est trouvée, ce programme synthétique peut être rapidement transformé en un programme réel. Cette transformation est faite en remplaçant les boucles vides par le code de l'application. A long terme, cette approche peut être adaptée à un environnement d'aide au développement de programmes parallèles. Par contre, cette méthode est plus coûteuse en mémoire par rapport à la méthode basée sur le noyau exécutif. L'émulation de la machine modélisée peut aussi être plus difficile, étant donné que certaines modifications doivent être réalisées directement sur le système d'exploitation. Le noyau exécutif permet lui une utilisation plus contrôlée de ce système d'exploitation.

### 5.2.4 L'analyse des performances

L'exécution de la charge synthétique est tracée : des informations de performance sont collectées **pendant** l'exécution. Le type et la quantité des données collectées dépendent des objectifs de l'analyse réalisée. Par exemple, le temps total d'exécution de la charge synthétique permet de constater que l'on va plus vite en parallèle. Les informations de charge des processeurs et des média de communication permettent une analyse plus ciblée sur la machine émulée : une distribution homogène de la charge représente une utilisation efficace de toutes les ressources disponibles (pas de sous-utilisation). Une analyse plus fine peut être réalisée si les traces sont collectées à chaque événement d'importance pendant l'exécution (e.g., un envoi de message, la fin de l'exécution d'un nœud de calcul). Toutefois, plus les traces sont détaillées, plus leur collecte est intrusive. Si la charge synthétique est exécutée par de vrais programmes synthétiques, alors les traces sont collectées par un moniteur standard disponible sur la machine cible. Si un noyau exécutif est utilisé, c'est le noyau lui-même qui éventuellement fait la collecte. Finalement, un outil de visualisation peut être employé afin d'aider à l'analyse des performances (e.g., ParaGraph [HE91]).

## 5.3 La version Méganode de *ANDES-Synth*

La version courante de *ANDES-Synth* s'exécute sur le Méganode de l'IMAG (voir l'appendice A). Le modèle de programme parallèle est écrit en *ANDES-C*. Le modèle de machine est un fichier contenant :

1. le nombre de processeurs;
2. la puissance de calcul en fonction de la puissance de calcul du Méganode (machine cible);
3. le modèle de communication;
4. la topologie du réseau de communication;
5. la fonction de traduction des types de base (entier, réels) en nombre d'octets;
6. la fonction de traduction des opérations de base (opération entière, opération réelle) en nombre d'itérations vides;
7. le degré de multiprogrammation.

Les stratégies d'implémentation incorporées dans l'outil sont :

1. un algorithme de groupement (l'algorithme DSC - *Dominant Sequence Clustering* - de Pyrros, un outil de groupement et ordonnancement de tâches [YG92]);
2. quelques heuristiques de placement [BTV93] (des algorithmes gloutons et itératifs) qui déterminent, avant l'exécution d'un programme parallèle, quel processeur exécute quel groupe de tâches. Le chapitre 6 décrit plus en détail ces algorithmes.

A partir du modèle du programme parallèle, du modèle de la machine parallèle et des stratégies d'implémentation, le fichier `andes.data` est généré. Ce fichier indique où (le processeur choisi par les stratégies de placement) est exécuté quoi (la charge synthétique : la structure du *DG-ANDES*, la composition des groupes et les coûts de calcul et de communication). Cette étape est réalisée sur la station de travail hôte du Méganode.

Dans la version courante de *ANDES-Synth*, l'exécution de la charge synthétique est assurée par un noyau exécutif. Une version préliminaire basée sur l'exécution d'un vrai programme synthétique (écrit en OUF [Arr92]) avait été développée, mais la mémoire disponible sur chaque processeur (1 Mégaoctet) n'était pas assez grande pour contenir plus de 50 nœuds de calcul par processeur. Avec le noyau exécutif et dépendant de la structure du *DG-ANDES*, il est possible d'avoir plus d'un millier de nœuds de calcul sur un processeur. C'est le noyau qui collecte les traces de l'exécution. Ces traces sont enregistrées dans un fichier qui est traité après la fin de l'exécution de

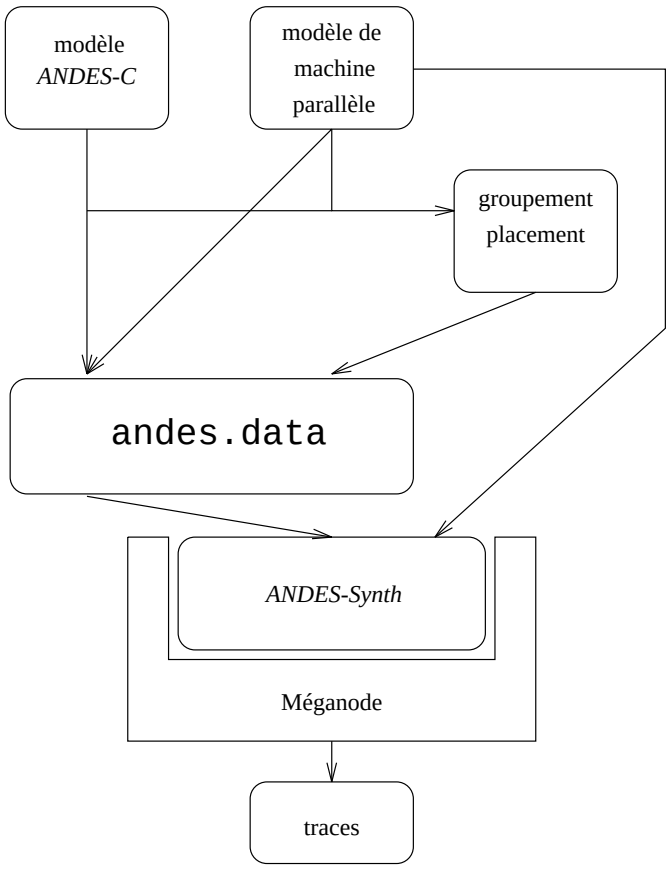



Figure 5.2 : La version Méganode de *ANDES-Synth*.

la charge synthétique. La Figure 5.2 présente la structure de la version Méganode de *ANDES-Synth* (à comparer avec la Figure 5.1).

Le noyau de *ANDES-Synth* est le *EM* (le gestionnaire d'exécution, en anglais *Execution Manager*) qui s'exécute sur chaque processeur du Méganode. Juste avant le démarrage du noyau, le Méganode est reconfiguré dans la topologie définie dans le modèle de machine (si cette topologie a un degré égal ou inférieur à 4). La simulation de topologies plus générales est prévue dans une version ultérieure de *ANDES-Synth*.

Après la reconfiguration, le noyau-exécutif :

1. lit le fichier `andes.data` qui donne les informations relatives au *DG-ANDES* (précédence, coûts de calcul et de communication, le placement choisi);
2. exécute la charge synthétique correspondante au fichier `andes.data`;
3. récupère les données nécessaires à l'analyse des performances de cette exécution (Figure 5.3).

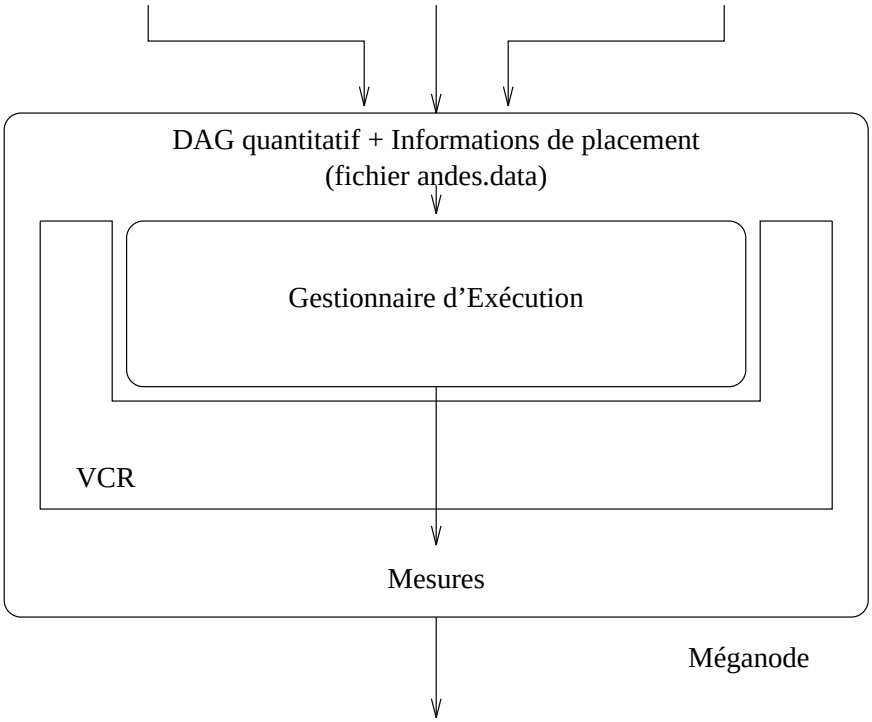



Figure 5.3 : Schéma général du gestionnaire d'exécution.

### 5.3.1 La structure du gestionnaire

Le gestionnaire est un programme parallèle SPMD (Programme Unique, Données Multiples - en anglais *Single Program, Multiple Data*, c'est-à-dire que le même code s'exécute sur chaque processeur du réseau) écrit en Inmos C parallèle (une version de C pour la construction de processus communicants). Le routage des messages entre les Transputers est assuré par VCR (*Virtual Channel Router*), un logiciel de routage développé à l'Université de Southampton, UK. Il y a deux types de processus (Figure 5.4) : un processus s'exécutant sur le Transputer interface entre l'hôte et le multiprocesseur (la tâche `root`) et ceux s'exécutant sur chaque processeur du réseau (les tâches `manager`). Par exemple, dans une grille comportant 16 processeurs, il y a 16 tâches `manager` et une tâche `root`. Ces tâches (toutes les tâches `manager` et la tâche `root`) sont logiquement complètement connectées : pour toutes tâches $t_i$ et $t_j$, il y a un canal logique allant de $t_i$ à $t_j$ (même si $i = j$).

### 5.3.2 Le processus `root`

La tâche `root` lit un fichier contenant la description du *DG-ANDES* et le placement statique choisi (le fichier `andes.data`). Elle envoie alors à chaque tâche `manager`

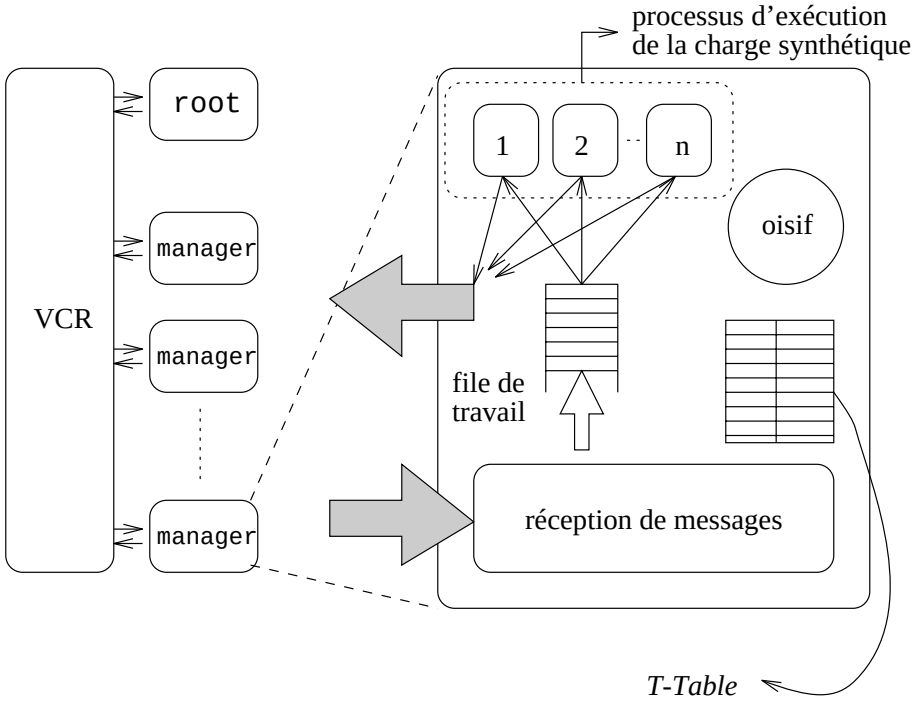



Figure 5.4 : Structure des tâches du gestionnaire.

$m_i$ les informations des nœuds de calcul placés sur le processeur de $m_i$. Ainsi, chaque `manager` a une connaissance partielle du *DG-ANDES*. De son côté, `root` garde les identificateurs des nœuds de calcul qui n'ont pas de prédécesseurs. Après le chargement des données du *DG-ANDES*, la tâche `root` envoie un signal aux nœuds de calcul racine (sans prédécesseurs) afin de démarrer l'exécution de la charge synthétique. Le temps de cette exécution est mesuré. Après l'envoi de ce signal de démarrage, `root` attend un signal de fin de l'exécution. Ce signal est envoyé par chaque tâche `manager`. A la fin de la réception de ce signal, `root` arrête de mesurer le temps. Finalement, les informations concernant la charge de chaque processeur sont envoyées par chaque `manager` à `root`.

### 5.3.3 Le processus `manager`

Chaque processus `manager` a deux rôles. Le premier est de recevoir, de la tâche `root`, les informations des nœuds de calcul du *DG-ANDES* gérés par le `manager` en question. Ces informations sont conservées dans une table (la *T-Table*). Le deuxième est de gérer effectivement l'exécution de la charge synthétique. Cette phase démarre seulement s'il y a des nœuds de calcul placés sur le processeur.

Trois types de (sous)processus sont crées avant la gestion de l'exécution de la

charge synthétique (Figure 5.4) :

1. un **processus de réception** : ce processus reçoit **tous** les messages destinés aux nœuds de calcul résidants dans le processeur. Lorsqu'un message arrive, ce processus de réception vérifie l'identificateur du nœud de calcul récepteur. Par exemple, le message est destiné au nœud de calcul $i$. La table *T-Table* est consultée afin de vérifier si tous les messages destinés à $i$ sont arrivés. Sinon, le compteur de messages arrivants de $i$ est mis à jour et le processus se met dans l'attente d'un nouveau message. Si tous les messages de $i$ sont arrivés, l'exécution de la charge synthétique correspondante au nœud de calcul $i$ peut démarrer. C'est le **processus d'exécution de la charge synthétique** qui démarre cette exécution. La communication entre ce processus et le processus de réception est assurée par une file de nœuds de calcul à être exécutés;

2. un **processus d'exécution de la charge synthétique** : lorsqu'un nœud de calcul est prêt à être exécuté, le processus de réception signale qu'il y a du travail à faire. Le premier processus libre (il peut y en avoir plusieurs) d'exécution de la charge synthétique consulte la file de nœuds de calcul prêts et exécute la charge correspondante à ce nœud. Cette exécution consiste à réaliser une boucle vide dont le nombre d'itérations est réglé par le coût de calcul du nœud. A la fin de cette exécution, tous les messages sont envoyés aux nœuds de calcul successeurs. Le nombre de processus d'exécution de la charge synthétique est réglé par le paramètre "degré de multiprogrammation" (paramètre présent dans le modèle de machine). Plus grand est ce degré, moins le processeur reste inactif (le Transputer permet le partage du temps entre plusieurs processus actifs);

3. un **processus oisif** : ce processus s'exécute seulement lorsqu'aucun autre processus (processus de réception, processus d'exécution de la charge synthétique ou processus de routage VCR) ne s'exécute. Ce processus oisif incrémente un compteur : la valeur de ce compteur est utilisée après l'exécution de la charge synthétique pour estimer la durée pendant laquelle le processeur a été oisif.

Le processus de réception de messages et les processus chargés de l'exécution de la charge synthétique s'exécutent en haute priorité, comme tous les processus de routage. L'exécution en haute priorité est interrompue seulement lors de l'attente de la réalisation d'une communication. Seul le processus oisif s'exécute en basse priorité, c'est-à-dire, qu'il se met forcément à la fin de la file de processus actifs après un intervalle fixe de temps. Le processus de réception de messages accepte une communication, consulte la *T-Table* (en utilisant un accès direct : aucune recherche séquentielle ou binaire n'est effectuée) et met un pointeur dans une file d'attente si un nœud de calcul doit être exécuté. Ces instructions sont exécutées assez vite et ce processus de réception est vite re-ordonnancé (en anglais *rescheduled*) dans l'attente d'un nouveau message. Par contre, il faut faire attention à l'exécution de la charge synthétique, car elle est faite en haute priorité et *a priori* elle n'est pas interrompue automatiquement. Afin

qu'une exécution synthétique ne monopolise pas le processeur, des instructions de re-ordonnancement sont mises dans les boucles vides. En d'autres mots, à chaque $x$ itérations de la boucle vide, le processus se met à la fin de la file d'attente de processus actifs. Le paramètre $x$ peut être modifié, mais un bon choix d'interruption correspond à la période d'exécution non-interrompue d'un processus en basse priorité (environ, un millier d'itérations). Ainsi, cette interruption forcée permet que d'autres processus s'exécutent, notamment les processus de routage.

Le processus `manager` s'arrête lorsqu'il n'y a plus de nœuds de calcul du *DG-ANDES* à être exécutés. Tous les processus (sauf les processus de routage) terminent aussi. Un message est envoyé au processus `root` pour signaler cette fin. Finalement, la valeur du compteur du processus oisif est envoyée au `root` afin d'être stockée pour un traitement *post-mortem*.

### 5.3.4 Traitement des informations des performances

Après la fin du gestionnaire de l'exécution synthétique, les données sur les performances relatives à cette exécution sont stockées dans la machine hôte. Pour chaque processeur, il est possible de calculer le temps pendant lequel ce processeur a été oisif ($tidle_p$). Le calcul est fait à partir de la valeur du compteur du processus oisif de chaque processeur. La valeur est donnée par un modèle linéaire :

$$tidle_p = \alpha + (\beta * n_{iter})$$

où $n_{iter}$ est le nombre d'itérations (la valeur du compteur). $\alpha$ et $\beta$ sont des constantes obtenues à partir du modèle de la machine parallèle. Le temps de l'exécution synthétique ($T_{ex}$) est aussi mesuré. Ainsi, le temps de travail d'un processeur $p$ ($twork_p$) est donné par :

$$twork_p = T_{ex} - tidle_p$$

Le temps $twork_p$ est décomposé en deux parties :

1. le temps passé à l'exécution des boucles vides correspondantes aux coûts de calcul des nœuds de calcul ($tap_p$);

2. le temps passé à :

   - l'exécution du processus de réception;
   - l'exécution d'une partie des processus responsables de l'exécution de la charge synthétique;
   - l'exécution des tâches de routage.

Cette deuxième fraction de $twork_p$ correspond au temps de surcharge de la machine (en anglais *overhead time* - $toh_p$) :

$$toh_p = T_{ex} - tap_p - tidle_p$$

pour chaque processeur $p$.

$tap_p$ est calculé à partir des coûts de calcul (exprimés en nombre d'itérations vides) des nœuds de calcul placés et de l'expression analytique du temps d'exécution en fonction des itérations vides. Cette expression est décrite dans le modèle de machine.

A partir de mesures réalisées sur l'outil, nous avons estimé le temps nécessaire pour gérer une exécution d'une charge synthétique quelconque ($OH$). De cette façon, on peut estimer la fraction (de $toh_p$) correspondante au temps dépensé par l'outil *ANDES-Synth*. Cette fraction est variable pour chaque processeur et dépend du nombre de messages qui arrivent ($MSGA$) et qui partent ($MSGD$), du nombre de nœuds de calcul placés sur le processeur ($NT$) et du degré de multiprogrammation ($MD$) :

$$OH = (104 * MSGA) + (204 * NT) + (215 * MD) + (40 * MSGD) + 80$$

Cette expression du calcul de la surcharge (en microsecondes) change selon l'architecture cible d'exécution de l'outil, mais les paramètres restent les mêmes ($MSGA$, $MSGD$, $NT$ et $MD$).

### 5.3.5 L'interface

Actuellement, l'interface de *ANDES-Synth* est construite sur l'interface graphique *X-Windows* pour les systèmes d'exploitation Unix. La Figure 5.5 présente le menu d'utilisation de *ANDES-Synth*. Le premier bouton (`Exper.data`) permet à l'utilisateur d'entrer le nom du modèle *ANDES*, le modèle de machine, la stratégie d'implémentation et le nombre d'exécutions de l'application (afin de constituer un espace d'échantillonnage). Le deuxième, le troisième et le quatrième boutons affichent respectivement le modèle *ANDES-C*, le modèle de machine et les stratégies d'implémentation employées. Le bouton `Go` démarre le traitement des modèles, l'utilisation des stratégies d'implémentation et l'exécution de la charge synthétique. Le bouton `Results` démarre le traitement de traces. Le bouton `See exper.` affiche les paramètres de l'expérience courante.

## 5.4 *OLGA* et *ANDES-Synth*

En général, les environnements de programmation parallèle (comme par exemple, PPSE [LER92]) peuvent être utilisés pour la génération de programmes synthétiques. Au lieu d'écrire un vrai programme, on écrit un programme synthétique. Le désavantage de cette approche est que la génération de ce programme synthétique n'est pas automatique. *ANDES-Synth* est un exemple d'outil orienté aux programmes synthétiques. A titre de comparaison, nous présentons un autre outil : *OLGA*.

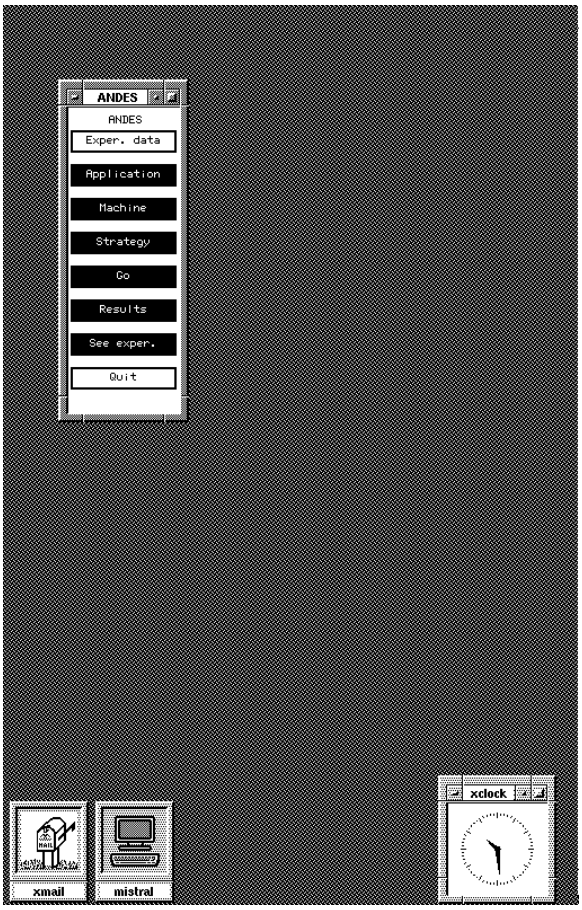



Figure 5.5 : Menu de *ANDES-Synth*.

*OLGA* (*Occam Load Generation Application* - génération en Occam de la charge de l'application) [WB93] est un environnement pour l'évaluation des performances de programmes synthétiques obtenus à partir d'un "squelette" (la structure des tâches synthétiques, c'est-à-dire, la description de l'ordre des phases de calcul et de communication) et à partir d'un fichier de paramètres qui décrit le type et la durée des phases de calcul et les tailles des données échangées entre les processus. Le squelette et le fichier de paramètres correspondent au modèle de programme *ANDES-C* de notre outil. Le squelette est un processus Occam qui contient le code à être exécuté sur chaque processeur. Il est obtenu à partir d'une bibliothèque de squelettes classifiés selon *BACS* (*Basel Algorithm Classification Scheme*), un schéma de classification d'algorithmes. *BACS* permet la description et la classification d'algorithmes parallèles. Par exemple, le programme ci-dessous est une partie d'un squelette associé à l'algorithme de Warshall (cet algorithme est composé de 2 étapes : une diffusion et un calcul) :

```
(...)
PROC skeleton (...)
  SEQ
    (... quelques initialisations...)
    -- tous les processus synthetiques synchronisent et
    -- commencent a mesurer le temps
    init.load(from.load,to.load)
    SEQ i=0 FOR problem.size
      SEQ
        -- decision si envoyer ou recevoir
        L.sender := my.turn(process.id,i)
        -- diffusion
        broadcast(L.sender,length,vector,
                  net.info,in.chs,out.chs)
        -- calcul
        run.load(load.id,from.load,to.load)
    -- fin de la prise de temps
:
```

Un fichier de paramètres de l'expérience est composé de 4 parties : `NODES` spécifie le nombre de processeurs actifs, `STRUCTURE` définit la topologie de processeurs, `WORK` décrit la charge de travail (e.g., nombre de calculs et tailles de communication) et `SKELETON` spécifie le squelette à utiliser. Les paramètres `NODES` et `STRUCTURE` correspondent à une version simplifiée du modèle de machine de *ANDES-Synth*.

L'environnement *OLGA* est ainsi composé de 3 parties :

- le *parser* lit le fichier de paramètres afin de déterminer la charge de travail;
- la *bibliothèque* contient les squelettes des algorithmes, les structures d'interaction entre les processus, les structures des données de base, les générateurs de nombres aléatoires, les primitives de chargement de processus et d'autres services globaux;
- le *chargeur de code* et *le frame* s'exécutent sur chaque processeur. Le *frame* est responsable de la mesure du temps et de l'exécution des charges de calcul et de communication. Il s'agit d'un noyau exécutif comme le *EM* dans *ANDES-Synth*.

*OLGA* s'exécute actuellement sur un réseau de Transputers. Cet environnement est utilisé pour l'évaluation de ParStone, un jeu de test synthétique basé sur *BACS* [WB93].

*ANDES-Synth* est un outil plus générique que *OLGA*. Le paradigme de programmation de *OLGA* est basé sur les processus séquentiels communicants, caractéristiques de l'environnement de programmation des Transputers. Le langage de description de modèles de programmes est Occam. Dans *ANDES-Synth*, on utilise C plus une bibliothèque. Cette bibliothèque peut permettre la modélisation de programmes qui suivent

d'autres paradigmes de programmation. Le modèle de machine décrit dans *OLGA* est aussi orienté vers la machine cible disponible. Il n'y a pas un souci de permettre l'émulation d'autres architectures que celle de la machine cible. La seule flexibilité est la possibilité de choisir différentes topologies, grâce à la reconfigurabilité de la machine cible. *OLGA* n'est pas orienté vers l'évaluation des stratégies automatisées d'implémentation. Le placement est défini par l'utilisateur. Le noyau exécutif des deux outils sont similaires, mais, dans *OLGA*, on est intéressé seulement par le temps d'exécution de la charge synthétique.

## 5.5 Conclusions

Ce chapitre a décrit l'outil *ANDES-Synth* dont la version courante s'exécute sur un multiprocesseur basé sur des Transputers. L'approche choisie est l'exécution d'une charge synthétique, donnant des informations plus proches d'une implémentation réelle du problème que celles données par d'autres techniques d'évaluation des performances comme la modélisation analytique et la simulation. L'outil est modulaire de façon à permettre l'évaluation de différents programmes parallèles, de différentes machines et de différentes stratégies d'implémentation.

Nous avons présenté *ANDES*, une technique de modélisation quantitative d'algorithmes parallèles. *ANDES-C* est sa forme d'expression. *ANDES-Synth* est un outil d'évaluation des performances de programmes synthétiques, générés à partir d'un modèle décrit en *ANDES-C*. Afin de clore la description de notre travail, le chapitre suivant présente un exemple d'application de l'outil *ANDES-Synth*. Cet outil est utilisé dans l'évaluation des stratégies de placement statique développées au sein de notre groupe de recherche.

# Chapitre 6

# Evaluation des Stratégies de Placement

Dans ce chapitre, nous présentons un exemple d'étude d'évaluation des performances en utilisant *ANDES-Synth*. Nous avons choisi la comparaison de stratégies de placement statique de tâches à cause de l'importance du placement lorsqu'on implémente un algorithme parallèle. Les stratégies évaluées ont été développées au sein du groupe Calcul Parallèle de l'IMAG [Bou94]. Tout d'abord, nous justifions l'importance du placement et de ses activités "cousines" : l'ordonnancement et le regroupement. Ensuite, le contexte des expériences est présenté : les modèles de machine et de programme employés, les stratégies de placement évaluées, le type de regroupement réalisé, les paramètres de l'expérience et le type d'analyse de données utilisés. Les résultats des expériences et les conclusions tirées sont présentés à la fin du chapitre.

## 6.1 Le programme parallèle et son implémentation

Notre point de départ est la chaîne d'implémentation d'un programme parallèle proposée dans [Bou94] (Figure 6.1). Cette chaîne représente la démarche couramment adoptée pour la transformation d'un programme écrit dans un langage source en un programme prêt pour être chargé et exécuté sur une machine parallèle.

Les textes source des tâches peuvent être écrits directement par le programmeur ou peuvent être dérivés à partir des programmes écrits en langages de plus haut niveau ou à partir de la parallélisation automatique de programmes séquentiels. La chaîne d'implémentation est composée de deux parties : la compilation des textes source des tâches (partie **A** dans la Figure 6.1), l'ordonnancement et le placement de tâches (partie **B** dans la Figure 6.1).

La **compilation** des textes source des tâches est réalisée de façon analogue à la compilation des programmes séquentiels. Le code pour la gestion du parallélisme (e.g., primitives de communication et de contrôle d'accès aux ressources partagées) est fourni par des bibliothèques spéciales (e.g., PVM [SGDM94] – voir l'arc (a) dans

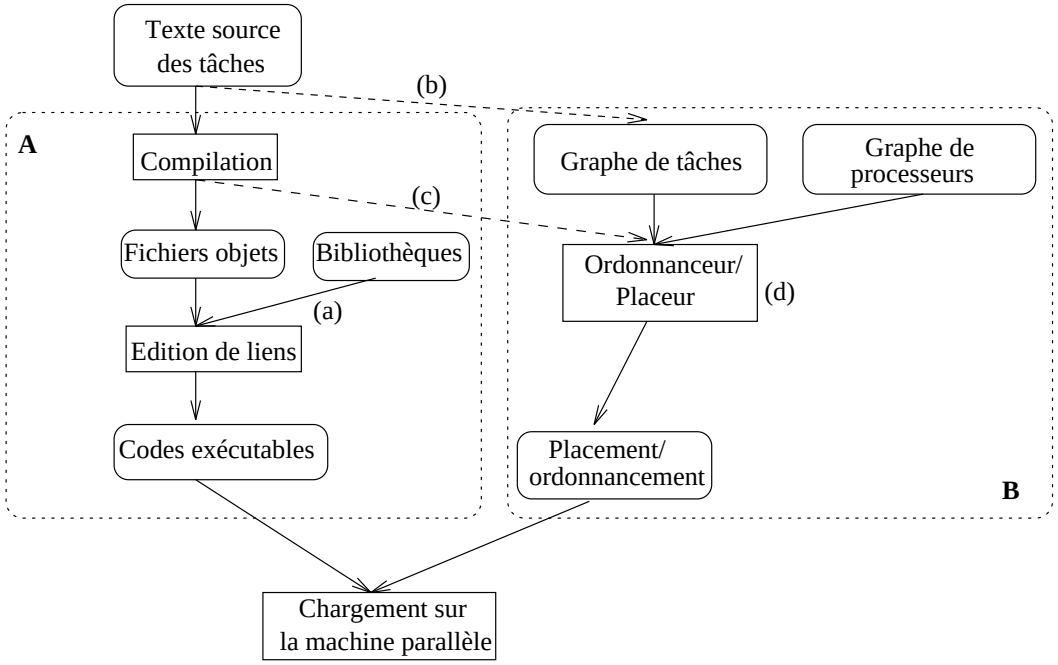



Figure 6.1 : La chaîne d'implémentation d'un programme parallèle [Bou94].

la Figure 6.1) qui sont utilisées lors de l'édition de liens afin de générer les codes exécutables.

L'**ordonnancement** et le **placement** définissent un choix d'allocation des ressources de calcul de la machine (les processeurs) aux tâches du programme. Dans le cas de l'ordonnancement, une date de début d'exécution est aussi calculée pour chaque tâche. Les entrées de l'ordonnancement et du placement sont un graphe des tâches du programme parallèle (voir les paragraphes 2.1 et 2.2) et un graphe de processeurs de la machine cible. Le graphe du programme est pondéré par des coûts qui modélisent les besoins de calcul, de communication et de mémoire. Le graphe de tâches peut être obtenu à partir d'annotations du programmeur dans les textes source des tâches (l'arc (b) dans la Figure 6.1) ou à partir de la compilation des tâches (l'arc (c) dans la Figure 6.1). Les informations fournies par les graphes sont essentielles pour déterminer le type d'algorithme d'allocation : un algorithme d'ordonnancement ou un algorithme de placement. Si la précédence est modélisée, alors le graphe de tâches est orienté (voir le chapitre 2) et une stratégie d'ordonnancement est employée. Pour chaque tâche, un processeur et une date de début d'exécution sont calculés. Si la précédence n'est pas représentée (i.e., le graphe n'est pas orienté), alors on utilise des algorithmes de placement. Dans ce cas, un processeur d'exécution est choisi pour chaque tâche. Un modèle de programme avec précédence peut être transformé en un modèle sans précédence, si un **regroupement** (en anglais *clustering*) de tâches est réalisé. Le graphe de

groupes est ensuite donné au placement. Le regroupement peut être considéré comme un ordonnancement où la machine parallèle cible est composée d'un nombre illimité de processeurs, tous connectés entre eux. Le placement consiste à se ramener à une architecture réelle (nombre limité de processeurs et pas toujours complètement connectés). Le schéma d'utilisation de l'ordonnancement, du regroupement et du placement est présenté dans la Figure 6.2 qui correspond au détail du module (d) de la Figure 6.1.

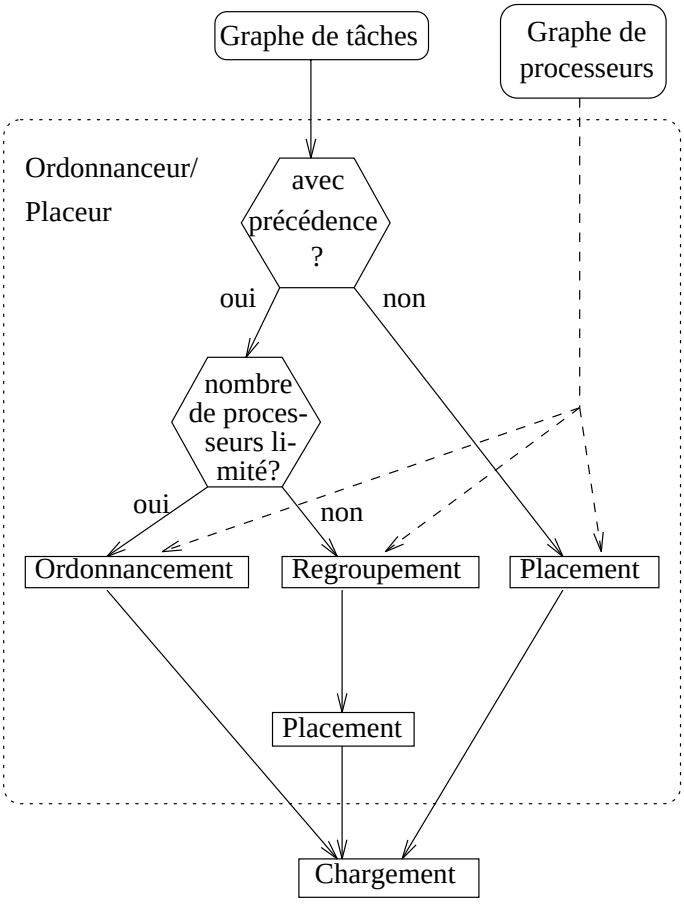



Figure 6.2 : L'ordonnancement, le regroupement et le placement [Bou94].

A la fin de la chaîne de la Figure 6.1, le chargement sur la machine parallèle est effectué en prenant les codes générés par la compilation. Le processeur et l'ordre d'exécution sont donnés par le placement et par l'ordonnancement. Regardons ensuite plus attentivement les algorithmes de regroupement et de placement, car la version courante de *ANDES-Synth* contient quelques algorithmes de ce type. Afin d'être complet, nous parlons brièvement du problème de l'ordonnancement.

### 6.1.1 L'ordonnancement et le regroupement

**L'ordonnancement [Bou94]**. Dans l'ordonnancement un objectif couramment utilisé est de minimiser le temps total d'exécution du programme parallèle. Il y peut avoir des sous-objectifs associés à cette minimisation comme, par exemple, la répartition homogène de la charge de calcul entre les processeurs. Les **algorithmes de liste** pour l'ordonnancement sont très utilisés. Ces algorithmes consistent à établir tout d'abord une liste de priorité des tâches suivant certains critères (e.g., coût de calcul décroissant) et ensuite à choisir les processeurs d'exécution suivant l'ordre donné par cette liste. Les relations de précédence inter-tâches doivent être respectées. Afin d'avoir une idée de la complexité du problème de l'ordonnancement, pour les graphes orientés avec un coût de calcul identique pour toutes les tâches et pas de coût de communication, il y existe un algorithme optimal d'ordonnancement d'un graphe de topologie arbitraire sur 2 processeurs. Cet algorithme a une complexité polynomiale. A nombre de processeurs fixé (> 2), on ne sait pas si le problème de l'ordonnancement est NP-difficile ou non. Pour des coûts de calcul non uniformes, le problème pour le cas général (c'est-à-dire, pour un graphe de topologie arbitraire) est NP-difficile. De nombreux travaux sur l'ordonnancement existent, principalement pour décrire des nouvelles heuristiques qui nous amènent de façon rapide à des solutions presque optimales.

**Le regroupement [GY92]**. Le regroupement consiste à grouper les tâches modélisées par un graphe orienté sans circuit. Toutes les tâches d'un groupe s'exécutent sur un même processeur. Dans un problème de regroupement, le modèle de machine parallèle considéré est un graphe complètement connecté avec un nombre illimité de processeurs. Un objectif couramment utilisé (comme pour l'ordonnancement) est de minimiser le temps de l'exécution parallèle : avec cet objectif, le regroupement est un problème NP-difficile (preuves de Chretienne, Papadimitrou & Yannakakis et Sarkar, cités par [GY92]). Un regroupement est **non-linéaire** si deux tâches indépendantes (cf. ci-dessous la définition de tâche indépendante) sont groupées dans le même groupe. Si ce n'est pas le cas, le regroupement est dit **linéaire**. Deux tâches sont indépendantes si elles n'ont pas une relation de précédence (directe ou non). En d'autres termes, si deux tâches sont indépendantes, alors il n'y a pas de chemin entre elles. Soit dans l'ordonnancement, soit dans le regroupement, deux tâches qui communiquent et qui sont placées sur le même processeur ont un coût de communication nul (car, sur un même processeur, on considère que seule l'adresse de la donnée est communiquée).

Les algorithmes de regroupement sont en général itératifs. Au début du regroupement, chaque tâche du graphe orienté est un groupe. Chaque itération consiste à regrouper les groupes formés dans l'itération précédente. Le critère de regroupement est défini par une fonction de coût. Pour limiter de façon polynomiale le nombre d'itérations par rapport au nombre de tâches du graphe orienté, les groupes formés dans une itération ne peuvent pas être décomposés dans une étape ultérieure (les algorithmes ne peuvent pas faire retour en arrière - en anglais *backtracking*). [GY92] donne 4 idées importantes pour comprendre le principe des algorithmes de regroupement :

1. lorsque 2 groupes sont regroupés, un ordonnancement doit être fait afin de

calculer le temps d'exécution correspondant au nouveau regroupement;

2. le regroupement de deux groupes correspond à annuler le coût de communication existant entre ces deux groupes;

3. le regroupement de deux groupes qui ne communiquent pas ne peut pas réduire le temps d'exécution. Au contraire, ce temps peut augmenter à cause de l'exécution séquentielle de tâches indépendantes;

4. si l'annulation du coût de communication entre deux groupes linéaires produit un groupe linéaire, alors ce regroupement n'augmente pas le temps d'exécution. Si le groupe produit est non-linéaire, la réduction du temps d'exécution dépend de la granularité du graphe orienté. En d'autres termes, la réduction aura lieu si les coûts de communications annulés par le regroupement étaient importants par rapport aux coûts de calcul des tâches du groupe formé.

Les idées ci-dessus montrent l'importance de l'annulation des coûts de communication dans le processus de regroupement. Les algorithmes basés sur l'annulation des coûts de communication forment une sous-classe importante des algorithmes de regroupement. Dans le paragraphe 6.2.5, nous décrivons un algorithme de regroupement linéaire basé sur l'annulation des coûts de communication dont le principe est la compression de la séquence dominante d'un graphe orienté (la **séquence dominante** est le plus long chemin en temps du graphe groupé et ordonnancé).

### 6.1.2 Le placement

Reprenons la définition formelle du problème de placement présentée dans le paragraphe 2.2. Un placement statique est une application *alloc* qui associe un processeur à une tâche avant l'exécution du programme parallèle [Bou94] :

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

où $T$ est l'ensemble des tâches à placer et $P$ l'ensemble de processeurs. Si $|P|$ est le nombre de processeurs et $|T|$ est le nombre de tâches, alors il existe $|P|^{|T|}$ placements possibles, ce qui rend trop coûteux l'exploration de toutes les possibilités de placement. Le problème du placement est NP-difficile [GJ79]. De façon similaire à l'ordonnancement et au regroupement, les objectifs du placement sont représentés par une fonction de coût. Une fonction de coût mesure la qualité du placement : elle est associée à une série de critères de qualité.

**Les fonctions de coût**. Considérons le cas où $|T| > |P|$, car c'est le plus fréquent en pratique. Pour $|T| \leq |P|$, le problème du placement peut se restreindre au problème du plongement [T+92]. Plusieurs fonctions de coûts peuvent être utilisées pour mesurer la qualité du placement. Dans le discours qui suit, $calc(t_i)$ correspond au coût de calcul d'une tâche (sa durée) et $comm(t_i, t_j)$ correspond à la durée de la communication de

la tâche $t_i$ vers la tâche $t_j$. Ces durées sont importantes (et dépendantes de la machine cible d'exécution), car un des objectifs est la minimisation du temps total d'exécution du programme parallèle. Par exemple, considérons une machine où les calculs ne sont pas réalisés en parallèle avec les communications (pas de recouvrement) et les communications d'une tâche ne sont pas effectuées simultanément. Dans telle machine, le **coût d'exécution** d'une tâche $t_i$ ($CE(t_i)$) pour un placement $alloc$ correspond à la durée de la tâche $t_i$ plus tous les coûts de communications qui partent de $t_i$ et qui ne sont pas des communications intra-processeur :

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

Le coût d'exécution cumulé sur un processeur $p_k$ est :

$$t_{alloc}(p_k) = \sum_{t_i|alloc(t_i)=p_k} [CE(t_i)]$$

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

Le coût $C_{alloc}$ du placement $alloc$ est le plus grand coût d'exécution de tous les processeurs :

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

Le coût du meilleur placement ($C^*$) est donc :

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

où $PL$ est l'ensemble de tous les placements possibles. La fonction de calcul ci-dessus est une approximation. L'hypothèse de base est de que toutes les tâches s'exécutent en parallèle. La précédence n'est pas considérée. La fonction de coût peut alors être raffinée afin de représenter mieux un comportement sur une modèle de machine spécifique :

- le recouvrement du calcul et des communications peut être modélisé par un coût d'exécution de la tâche $t_i$ basé sur un $max$ de la durée de $t_i$ et de la somme des coûts de communications inter-processeurs originaires de $t_i$;
- les communications d'une tâche $t_i$ peuvent être simultanées;
- les durées des tâches peuvent être modifiées selon le placement choisi : les processeurs de la machine parallèle peuvent avoir différentes puissances de calcul;

- la distance entre les processeurs peut être prise en compte lors du calcul du coût de communication entre deux tâches.

Chaque aspect présenté ci-dessus nous amène à une fonction de coût différente. Comme pour les stratégies d'ordonnancement et de regroupement, d'autres sous-objectifs peuvent être associés au but principal d'un placement : l'exemple typique est d'avoir comme le but principal la réduction du temps d'exécution et comme but secondaire l'équilibrage de la charge de calcul entre les processeurs de la machine parallèle.

**Les différentes solutions**. Les nombreux algorithmes de placement disponibles dans la littérature peuvent être classifiés en :

- **algorithmes exacts** : le principe de ces algorithmes repose sur une exploration de toutes les solutions possibles. La solution optimale est trouvée, mais de façon coûteuse à cause de l'explosion combinatoire. Une approche usuelle est l'exploration de l'arbre de construction de placements (la racine correspond à la situation où aucune tâche n'est placée et les feuilles correspondent à l'ensemble de tâches placées). Autre approche est la décomposition du problème en sous-problèmes. Une recherche arborescente (du type *Branch & Bound*[1]) peut être alors utilisée;

- **algorithmes heuristiques** : ces algorithmes conduisent à une solution proche de l'optimum. On en envisage deux catégories : les **algorithmes gloutons** permettent de construire à coût bas une solution en partant d'une solution partielle que l'on complète. Dans le cas du placement, la $q$-ième tâche est allouée à un processeur selon un critère à partir du placement partiel appliqué aux $(q-1)$ premières tâches. Les algorithmes de liste sont des algorithmes gloutons. Les **algorithmes itératifs** partent d'une solution complète (pas forcément très bonne – un algorithme glouton peut être utilisé pour déterminer une solution complète initiale) que l'on améliore par transformations élémentaires (en général, par des permutations de tâches). Dans les algorithmes itératifs, des perturbations aléatoires sont souvent nécessaires afin d'éviter les minima locaux pour les valeurs de la fonction de coût;

- **heuristiques obtenues comme généralisations d'algorithmes exacts** : ces algorithmes sont dérivés d'algorithmes exacts qui ne sont applicables que dans des cas restreints. Cette dérivation est assurée par la relaxation de certaines contraintes de ces algorithmes exacts.

### 6.1.3 L'ordonnancement et le regroupement versus le placement

Suivant le modèle de programmation, on sera amené à faire de l'ordonnancement, du regroupement ou du placement (cf. le chapitre 2). Par exemple, l'ordonnancement

[1] expression sans correspondante dans la langue française : il s'agit d'une méthode de recherche de solution par décomposition et bornage.

exige un modèle de programme où la précédence est représentée. Par contre, pour le placement, les modèles de programme n'ont pas besoin de contenir ce type d'information. La **granularité** du programme parallèle considéré lorsqu'on envisage d'utiliser des stratégies d'**ordonnancement** et de **regroupement** est généralement faible, ce qui amène un avantage et deux désavantages :

⊕ une solution plus fine et donc plus performante peut être obtenue;

⊖ un nombre important de tâches conduit à l'explosion combinatoire des possibilités d'ordonnancement. Seuls des algorithmes de faible complexité peuvent être utilisés de manière pratique;

⊖ dans ce genre de problème, les particularités de l'architecture cible ou du système d'exploitation ont une grande influence sur la solution. Le modèle de machine devrait donc être aussi précis que le modèle de programme.

Au contraire, la **granularité** des programmes soumis au **placement** est en général assez forte, ce qui amène deux avantages et un désavantage :

⊕ un nombre faible de tâches (ceci peut arriver souvent dans la réalité) 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 d'annuler l'influence de certaines particularités du système d'exploitation ou de la machine;

⊖ la solution envisagée ne prend pas en compte le comportement fin des programmes.

Par rapport au type d'information produit par ces opérations :

⊕ l'**ordonnancement** et le **regroupement** comportent une allocation de processeurs et une date de démarrage des différentes tâches;

⊖ le **placement** ne fournit qu'une allocation de processeurs.

Les problèmes d'ordonnancement, de regroupement et de placement analysés dans ce chapitre travaillent sur des programmes dont le comportement est connu de façon **statique** (à la compilation) :

⊖ un certain nombre de programmes parallèles ne peuvent pas être traités par ces solutions, par exemple, les programmes parallèles non-déterministes et les programmes avec la création dynamique de tâches.

Finalement :

⊖ les problèmes de l'ordonnancement, du regroupement et du placement sont NP-difficiles.

## 6.2 Le contexte expérimental

Comme on l'a cru dans les paragraphes précédents, le problème de l'allocation de tâches aux processeurs est un problème théoriquement difficile. Des solutions pratiques (par exemple, les heuristiques) existent, mais il est actuellement impossible de savoir théoriquement quelle est la meilleure. Nous essayons ici d'apporter une réponse expérimentale à cette question à l'aide de *ANDES-Synth* (l'outil présenté dans le chapitre 5). Les objectifs des expériences sont détaillés dans le paragraphe suivant. Ensuite, nous présentons les modèles de machine, d'algorithme et les stratégies d'allocation (2 stratégies de regroupement et 21 stratégies de placement) utilisés dans ce cadre expérimental. Finalement, nous dressons une liste des paramètres complémentaires de nos expériences.

### 6.2.1 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 **réels** 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 grâce à un indice de performance (le temps d'exécution) mesuré à partir de l'exécution d'une charge synthétique sur une machine parallèle **réelle**. Comme on l'a vu dans le chapitre 5, dans la durée d'une exécution réelle, trois types de temps existent : le temps d'exécution des tâches de calcul, le temps d'exécution des tâches de gestion des ressources (e.g., la communication et le système d'exploitation) et le temps d'inactivité des processeurs. 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.

Par opposition à ce temps **mesuré**, les fonctions de coût des stratégies de placement donnent un temps **théorique approximatif** d'exécution d'un programme parallèle sur une machine parallèle (les deux représentés par leurs modèles respectifs). Notre objectif est aussi de vérifier si les coûts d'exécution mesurés 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 de comprendre les raisons de la disparité des performances trouvées.

Finalement, comme conséquence de cette analyse, nous espérons obtenir un ensemble de "règles d'or" afin de guider le programmeur ou le compilateur (voir la Figure 6.1) à choisir la stratégie de placement adéquate selon les paramètres (quantitatifs et structuraux) du programme parallèle et de la machine parallèle.

Ce travail est étroitement lié au travail de thèse de Pascal Bouvry [Bou94] concernant le développement d'une plate-forme d'aide au placement.

### 6.2.2 Le modèle de machine

Le modèle de machine choisi pour nos expériences représente un ordinateur très proche du multiprocesseur cible sur lequel *ANDES-Synth* s'exécute (le Méganode – voir l'appendice A). Le modèle de machine représente alors 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 – Figure 6.3). 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 sont peu variables pour une taille de message donnée. Cela nous donne un modèle de communication linéaire simple et fiable. Le tore a été choisi à cause de son faible diamètre.

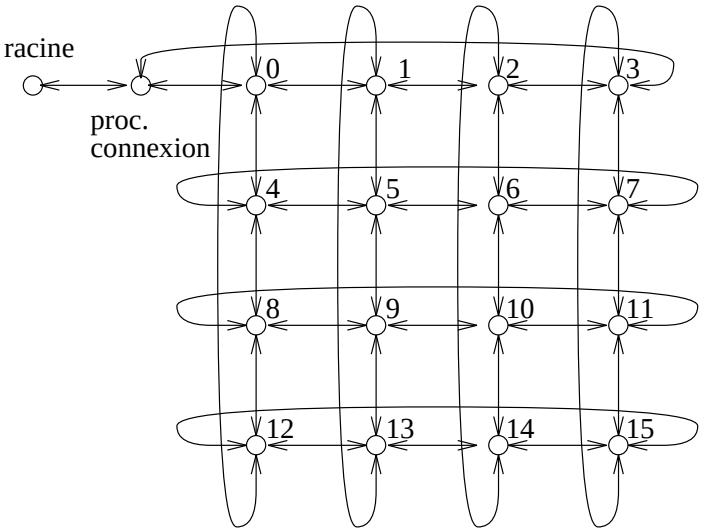



Figure 6.3 : La topologie spécifiée dans le modèle de machine.

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

$$t_{calc} = 1,76 * n_{ops}$$

Le processeur de base est multiprogrammé et on peut régler son degré de multiprogrammation, c'est-à-dire, on peut limiter le nombre de tâches de calcul traitées simultanément par le processeur. Dans nos expériences, nous disposons d'un modèle de machine sans multiprogrammation (degré 1 de multiprogrammation) et sans préemption : lorsqu'un nœud de calcul s'exécute, il le fait jusqu'à l'occurrence d'une

communication. Nous pouvons faire des expériences en relâchant ces contraintes : avec des degrés de multiprogrammation différents de 1 et un modèle avec préemption où, à chaque $x$ itérations vides, le processus de calcul se met à la fin de la file de processus actifs, $x$ étant un paramètre du modèle de machine. Il faut remarquer que l'absence de multiprogrammation dans le contexte de l'exécution d'un nœud de calcul ne représente pas une absence de multiprogrammation dans le contexte d'exécution d'un groupe. Finalement, les nombres 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 (en anglais *acknowledgement*) 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 (en microsecondes) :

$$t_{comm} = 188,4 + (1,0 * n_{octets}) \quad n_{octets} \leq 160$$

$$t_{comm} = 210,0 + (1,5 * n_{octets}) \quad n_{octets} > 160$$

Pour une communication à distance 2, on a :

$$t_{comm} = 297,2 + (2,2 * n_{octets}) \quad n_{octets} \leq 160$$

$$t_{comm} = 464,5 + (1,6 * n_{octets}) \quad n_{octets} > 160$$

Pour une communication à distance 3, on a :

$$t_{comm} = 358,9 + (3,2 * n_{octets}) \quad n_{octets} \leq 160$$

$$t_{comm} = 711,5 + (1,6 * n_{octets}) \quad n_{octets} > 160$$

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

$$t_{comm} = 438,6 + (4,4 * n_{octets}) \quad n_{octets} \leq 160$$

$$t_{comm} = 999,2 + (1,7 * n_{octets}) \quad n_{octets} > 160$$

Ces modèles ont été obtenus à partir des mesures réalisées sur le Méganode, configuré en tore 4x4, sans l'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 machine.

**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 machine donnés au placement représentent des processeurs avec une taille illimitée de mémoire (un modèle de machine avec une mémoire virtuelle).

Ce modèle de machine a été choisi très proche de la machine cible sur laquelle *ANDES-Synth* s'exécute. Ce choix a été fait par souci de simplicité et comme première approche. Dans l'avenir, nous émulerons d'autres machines différentes de la machine cible.

Une remarque importante est que l'outil *ANDES-Synth* représente lui-même une surcharge. Cette surcharge a été mesurée (voir le paragraphe 5.3.4). Le modèle de machine est alors modifié afin de que cette surcharge en temps soit plus petite qu'un pourcentage $y$ du temps total d'exécution mesuré. Pour nos expériences, on a considéré 10% de surcharge par rapport au temps d'exécution.

### 6.2.3 Les modèles d'algorithmes : le jeu de test

Il n'existe pas de jeu de test d'algorithmes parallèles standard et représentatif qui puisse être utilisé dans nos expériences. La représentativité est importante pour la généralisation des conclusions tirées à partir de comparaisons expérimentales. Nous avons alors développé un jeu de test avec 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 test de **programmes** parallèles pour les multiprocesseurs à mémoire distribuée et à échange de messages. Nous espérons que la représentativité de notre jeu de test est satisfaisante et nous essayerons de justifier nos choix. Dans l'avenir et selon les besoins, de nouveaux modèles pourront être ajoutés à notre jeu de test. Nous détaillons ensuite chacun de ses algorithmes.

`BELLFORD2`. L'algorithme de Bellman-Ford [BT89] résout le problème du plus court chemin entre tous les nœuds d'un graphe orienté pondéré et un nœud destination de ce graphe. Le nœud destination n'a pas de successeurs. Dans la Figure 6.4, la structure du *DG-ANDES* de `BELLFORD2` est présentée pour le graphe orienté `EXEMPLE` (la fonction des arcs pointillés dans le *DG-ANDES* est expliquée dans le paragraphe 6.2.5 – il s'agit des arcs du type `CONTEXT`). Tous les nœuds de calcul ont un coût de calcul de 10 opérations entières. Chaque communication du *DG-ANDES* a un poids de 1 entier. Tous les descripteurs d'entrée et de sortie sont du type `AND`.

`DIAMOND1`. Ce modèle représente un calcul systolique décrit dans [IS90] où le graphe est connu sous le nom de "graphe espace-temps" (Figure 6.5). Tous les nœuds de calcul ont un coût de calcul de 10 opérations entières. Chaque communication du *DG-ANDES* a un poids de 1 entier. Tous les descripteurs d'entrée et de sortie sont du type `AND`. Ces paramètres quantitatifs sont aussi valables pour les modèles `DIAMOND2`, `DIAMOND3` et `DIAMOND4`.

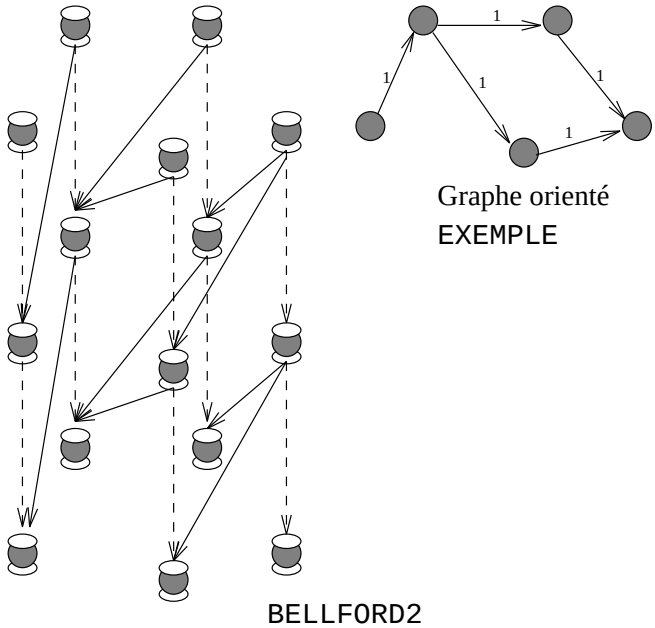



Figure 6.4 : La structure de BELLFORD2.

DIAMOND2. Ce modèle représente un calcul systolique de multiplication de matrices décrit dans [LK90] (Figure 6.5).

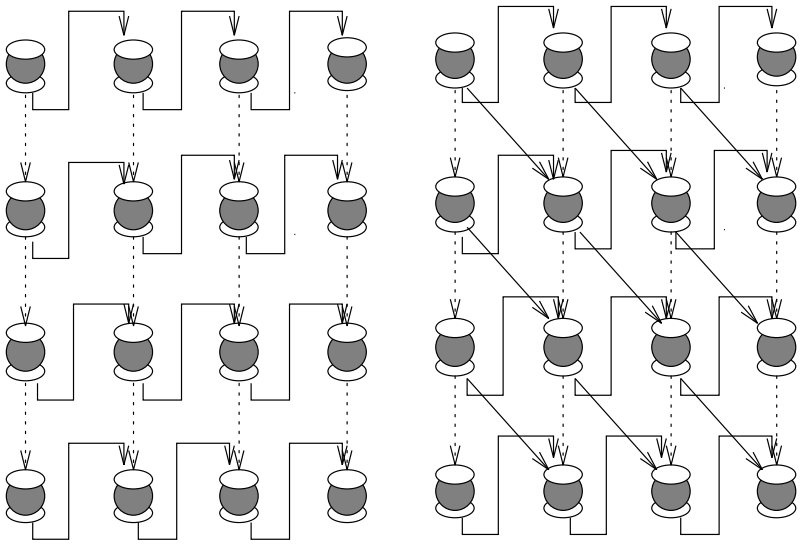
DIAMOND3. Ce modèle représente un calcul systolique de multiplication de matrices décrit dans [KCN90]. Ce modèle décrit un calcul de granularité plus fine pour la multiplication que celui modélisé par DIAMOND2 (Figure 6.6).

DIAMOND4. Ce modèle représente un calcul systolique de la fermeture transitive d'une relation sur un ensemble d'éléments [SC92] (Figure 6.7).

DIVCONQ. Ce modèle représente un algorithme du type diviser-pour-paralléliser [MS91] (Figure 6.8). Tous les nœuds de calcul ont un coût de calcul de 10 opérations entières. La première division représente une communication de 1024 entiers. A chaque nouvelle division, la taille des données communiquées est réduite à la moitié. Tous les descripteurs d'entrée et de sortie sont du type AND. Le modèle est construit à partir des appels récursifs à la procédure de division.

FFT2. Il s'agit d'une transformée rapide de Fourier unidimensionnelle. Ce modèle a été dérivé d'un programme de *GENESIS 2.2* (Figure 6.9). Tous les nœuds de calcul ont un coût de 100 opérations entières. Chaque communication du *DG-ANDES* a un poids de 1 réel. Tous les descripteurs d'entrée et de sortie sont du type AND.

GAUSS. Ce modèle correspond au graphe d'exécution de l'algorithme d'élimination de Gauss utilisé dans la résolution de systèmes linéaires [BT89][CT93] (Figure 4.4 dans le chapitre 4). Tous les nœuds de calcul ont un coût de calcul de 10 opérations entières. Chaque communication du *DG-ANDES* a un poids de 1 entier. Tous les descripteurs



DIAMOND1 (taille du problème : 4)

DIAMOND2 (taille du problème : 4)

Figure 6.5 : Les structures de DIAMOND1 et de DIAMOND2.

d'entrée et de sortie sont du type AND.

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 communication de données vers les nœuds de l'étage successeur. Le choix des nœuds récepteurs est aléatoire. Dans la Figure 6.10, nous présentons un exemple de ITERATIVE2. Tous les nœuds de calcul ont un coût de calcul de 10 opérations entières. Chaque communication du *DG-ANDES* a un poids de 256 entiers, sauf les communications entre les nœuds de calcul qui modélisent une composante de l'itération : la communication intra-composante a un poids de 1 entier. Tous les descripteurs d'entrée et de sortie sont du type AND.

MS3. Ce modèle correspond à une représentation générique d'un algorithme du type série-parallèle [MS91]. MS3 est plus détaillé que le graphe présenté dans [MS91], car la partie parallèle du calcul es composée d'un ensemble d'itérations (Figure 6.11). Tous les nœuds de calcul ont un coût de calcul proportionnel à la taille du problème traité. Chaque communication du *DG-ANDES* a aussi un poids proportionnel à la taille du problème traité. Tous les descripteurs d'entrée et de sortie sont du type AND.

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.

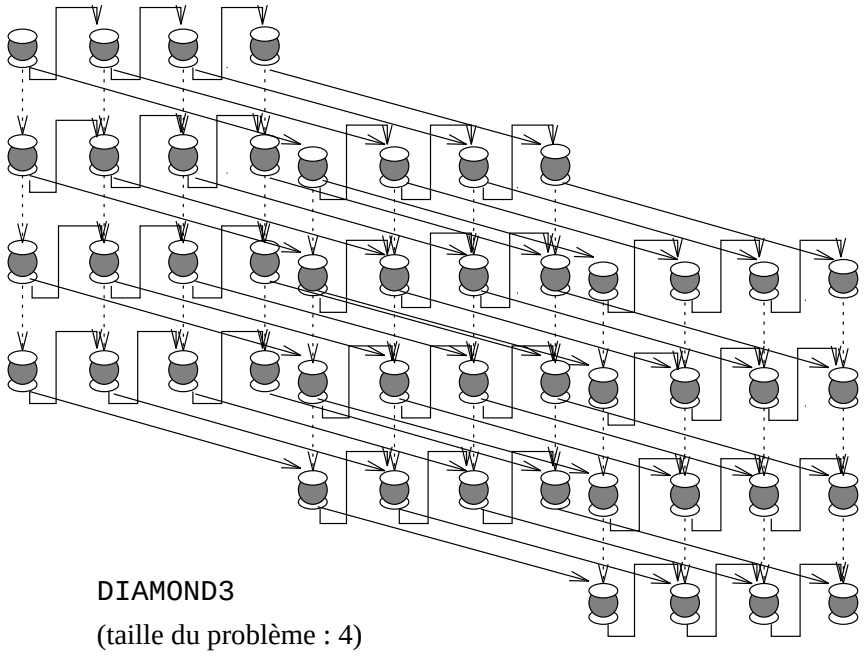



Figure 6.6 : La structure de DIAMOND3.

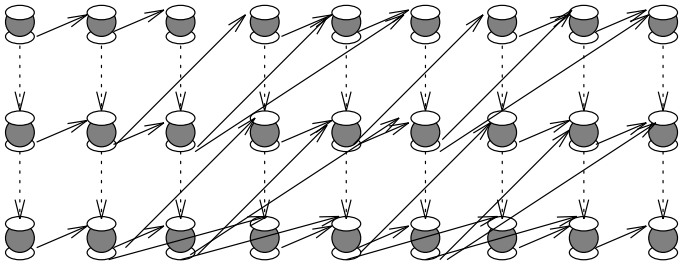
PDE1. Il s'agit du modèle d'un algorithme de résolution de l'équation de Poisson dans l'espace tridimensionnel. La méthode employée est la relaxation rouge-noir (en anglais *red-black relaxation*). Ce modèle a été dérivé du programme PDE1 de *GENESIS 2.2* (Figure 6.12). Tous les nœuds de calcul ont un coût de calcul de 10 opérations entières. Chaque communication du *DG-ANDES* a un poids de 1 entier. Tous les descripteurs d'entrée et de sortie sont du type AND. On trouve les mêmes caractéristiques dans PDE2 et dans QCD2.

PDE2. Modèle similaire à PDE1, mais concernant un espace bidimensionnel. La méthode de résolution de l'équation de Poisson est la multi-grille (en anglais *multi-grid*). PDE2 fait aussi partie du jeu de test de *GENESIS 2.2* (Figure 6.13).

PROLOG. Ce modèle est présenté dans le chapitre 4 (voir la Figure 4.6). Tous les nœuds de calcul ont un coût de calcul de 10 opérations entières. Chaque communication du *DG-ANDES* a un poids proportionnel à la taille du problème traité. Tous les descripteurs d'entrée et de sortie sont du type AND.

QCD2. Basé aussi sur un programme de *GENESIS 2.2*, QCD2 est le modèle *ANDES* d'un algorithme de résolution de systèmes linéaires. La méthode employée est le gradient conjugué [BT89] (Figure 6.14).

STRASSEN2. Ce modèle est présenté dans le chapitre 4 (voir la Figure 4.5). Tous les nœuds de calcul qui modélisent une multiplication ou une addition entre entiers ont un coût de calcul de 1. Les données traitées sont des entiers. Tous les descripteurs



DIAMOND4
(taille du problème : 3)

Figure 6.7 : La structure de DIAMOND4.

d'entrée et de sortie sont du type AND. STRASSEN2 est le modèle le plus fidèle à la réalité (par rapport aux coûts) entre tous les modèles de notre jeu de test.

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 jeu de test ParStone (lié au schéma de classification *BACS*). La structure du *DG-ANDES* est présentée dans la Figure 6.15. Tous les nœuds de calcul ont un coût de calcul proportionnel à la taille du problème. Chaque communication du *DG-ANDES* a un poids de 1 entier. Tous les descripteurs d'entrée et de sortie sont du type AND.

Les modèles ci-dessus représentent des algorithmes scientifiques très connus dans la littérature et dans la pratique. Ils couvrent aussi différents types de décomposition parallèle (régulier, diviser-pour-paralléliser, pipeline). De cette façon, nous croyons que notre choix du jeu de test est bon.

### 6.2.4 Paramètres du jeu de test - *DG-ANDES*

Nous présentons une caractérisation de notre jeu de test basée sur de paramètres structuraux que nous jugeons important dans un contexte d'évaluation des performances.

**Taille du problème**. Un aspect essentiel dans la présentation de notre jeu de test est la normalisation des tailles de problème entre les différents modèles. Nous avons alors supposé que tous les problèmes traitent une matrice de $N \times N$ éléments (dans nos expériences, $N = 32$). 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 (un programme logique est défini par un but à atteindre à partir d'un ensemble de faits) nous amène à un ensemble maximal de $N \times N$ possibilités. D'autres problèmes

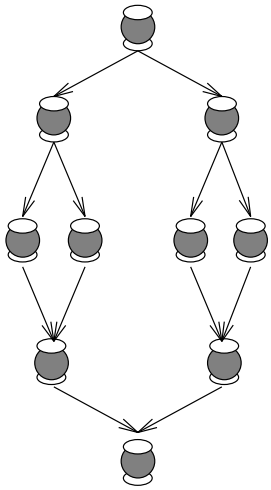



Figure 6.8 : La structure de `DIVCONQ`.

qui traitent des matrices avec $N \times N (N = 32)$ 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.

**Parallélisme virtuel (PV)**. On peut estimer le nombre moyen de nœuds de calcul qui s'exécutent simultanément en divisant le nombre total de nœuds de calcul par la longueur du chemin critique du *DG-ANDES*. Cette longueur n'est pas donnée en temps, mais en nombre de nœuds de calcul et elle correspond au nombre d'étages du *DG-ANDES*. Le parallélisme virtuel exprime une distance (par rapport à la structure du graphe) entre un programme séquentiel ($PV = 1$) et un programme parallèle (*nnc* – nombre de nœuds de calcul du *DG-ANDES*).

| modèle | ms3 | qcd02 | gauss | diam1 | diam2 | diam4 | msgas | bell2 | warsh |
|---|---|---|---|---|---|---|---|---|---|
| PV | 3,72 | 15,55 | 15,59 | 15,78 | 15,78 | 20,88 | 25,71 | 28,08 | 30,18 |
| *nnc* | 4062 | 1026 | 530 | 1026 | 1026 | 1002 | 3420 | 365 | 1026 |
| modèle | diam3 | prolg | divcq | stra2 | iter2 | pde02 | pde01 | fftr2 | |
| PV | 33,40 | 77,08 | 80,74 | 92,69 | 213,50 | 216,77 | 366,00 | 426,83 | |
| *nnc* | 1002 | 1002 | 1534 | 1483 | 2562 | 2818 | 5124 | 5122 | |

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 machine associé. Utilisons alors le modèle de machine présenté dans le paragraphe 6.2.2.

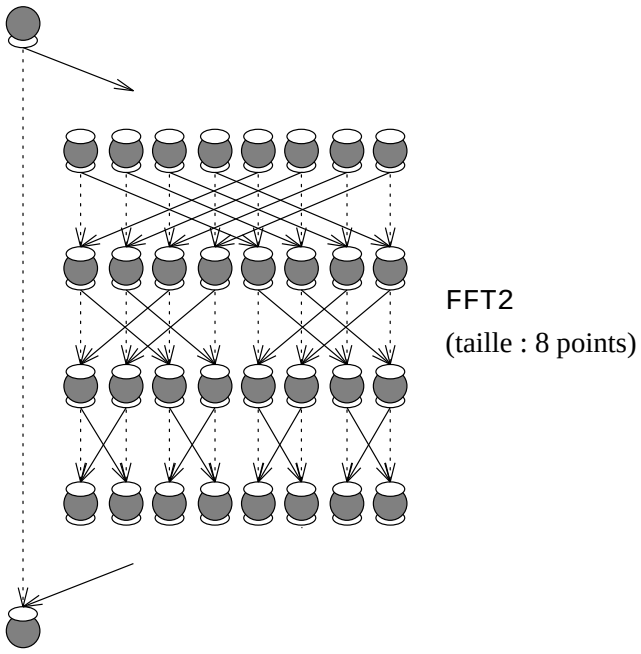



Figure 6.9 : La structure de FFT2.

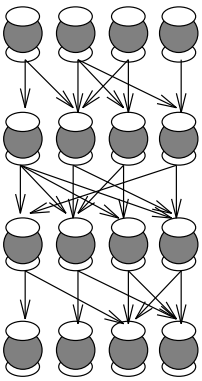
**Total estimé de temps de calcul**. Avec la vitesse de calcul du processeur, on peut estimer le temps de calcul nécessaire pour exécuter tous les nœuds du *DG-ANDES*. Les coûts de calcul (exprimés en nombre d'opérations entières) des nœuds de calcul sont d'abord transformés en opérations de base et puis convertis en temps estimés.

| modèle | warsh | bell2 | stra2 | diam4 | diam2 | gauss | iter2 | diam3 | prolg |
|---|---|---|---|---|---|---|---|---|---|
| secondes | 5,80 | 6,41 | 8,62 | 8,84 | 9,05 | 9,35 | 11,32 | 17,67 | 17,67 |
| modèle | diam1 | divcq | fftr2 | pde02 | pde01 | msgas | qcd02 | ms3 | |
| secondes | 18,09 | 27,05 | 45,19 | 49,70 | 54,24 | 164,03 | 180,85 | 183,33 | |

Selon l'ordre de grandeur des temps estimés de calcul, il y a trois groupes de modèles :

1. temps estimé plus petit que $10^6\mu$s :WARSHALL, BELLFORD2, STRASSEN2, DIAMOND4, DIAMOND2 et GAUSS;
2. temps estimé entre $10^6\mu$s et $10^7\mu$s : ITERATIVE2, DIAMOND3, PROLOG, DIAMOND1, DIVCONQ, FFT2, PDE2 et PDE1;
3. temps estimé plus grand que $10^7\mu$s : MS_GAUSS, QCD2 et MS3.

**Total estimé de temps de communication**. Avec les modèles de communication, on peut estimer tous les temps de communication du *DG-ANDES*. D'abord, il faut



ITERATIVE2
(4 composantes)

Figure 6.10 : La structure de ITERATIVE2.

convertir les entiers et réels communiqués en nombre d'octets. Ensuite, par souci de simplicité, nous utilisons le modèle pour les communications à distance 2 pour estimer les temps de communication. Cette distance correspond à la moitié de la distance plus longue du tore 4x4 (rappelons que le placement n'a pas été fait : alors on ne sait pas où les nœuds de calcul seront exécutés). Pour chaque programme test, nous avons modifié tous les coûts de communication 2 fois afin d'avoir un jeu de test plus grand. Les coûts de communication ont été multipliés par un facteur *a* ($a > 1$) et par un facteur *c* ($c < 1$). Le facteur $b = 1$ correspond aux coûts originaux. Nous n'avons pas utilisé le même facteur pour tous les tests à cause du besoin de garder une durée d'exécution approximativement 10 fois plus grande que la surcharge du gestionnaire de l'exécution synthétique. Avec 3 coûts de communication et 17 modèles du jeu de test, nous avons fait alors des expériences avec 17*3 modèles. La variation du coût de communication par chaque modèle est donnée par le Tableau 6.1 ($t_{msg}$ correspond à la taille de message).

Pour le cas où on a le facteur *a* de communication, on a :

| modèle | divcq | warsh | bell2 | diam4 | diam1 | diam3 | iter2 | gauss | pde02 |
|---|---|---|---|---|---|---|---|---|---|
| $\mu$s ($10^6$) | 2,60 | 5,12 | 9,10 | 11,84 | 13,94 | 20,02 | 23,61 | 34,09 | 55,69 |
| modèle | stra2 | fftr2 | prolg | pde01 | ms3 | diam2 | msgas | qcd02 | |
| $\mu$s ($10^6$) | 55,92 | 71,84 | 99,04 | 100,99 | 138,59 | 171,91 | 251,08 | 417,23 | |

Pour le cas où on a le facteur *b* de communication, on a :

| modèle | bell2 | divcq | diam1 | warsh | diam3 | gauss | diam4 | prolg | stra2 |
|---|---|---|---|---|---|---|---|---|---|
| $\mu$s ($10^6$) | 1,45 | 2,13 | 2,23 | 3,04 | 3,13 | 7,20 | 10,08 | 10,53 | 12,03 |
| modèle | iter2 | fftr2 | pde02 | pde01 | ms3 | diam2 | qcd02 | msgas | |
| $\mu$s ($10^6$) | 12,46 | 18,17 | 30,79 | 55,83 | 72,08 | 86,64 | 88,11 | 127,78 | |

Pour le cas où on a le facteur *c* de communication, on a :

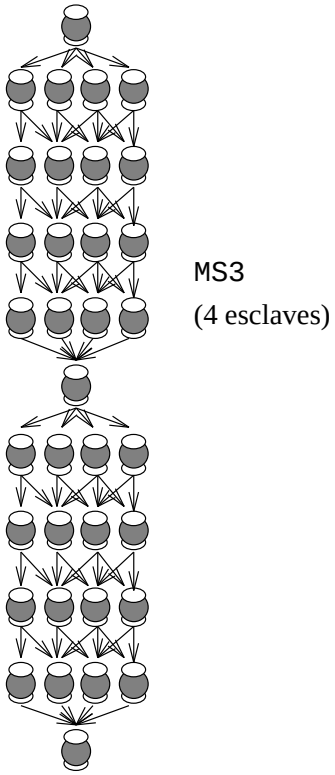



Figure 6.11 : La structure de MS3.

| modèle | bell2 | diam1 | gauss | diam3 | divcq | prolg | warsh | stra2 | diam4 |
|---|---|---|---|---|---|---|---|---|---|
| $\mu$s ($10^6$) | 0,69 | 1,05 | 1,15 | 1,44 | 1,66 | 1,68 | 2,00 | 2,16 | 3,02 |
| modèle | iter2 | diam2 | fftr2 | qcd02 | msgas | pde02 | ms3 | pde01 | |
| $\mu$s ($10^6$) | 3,54 | 9,90 | 11,47 | 14,06 | 16,81 | 18,34 | 18,87 | 33,25 | |

**Rapport communication/calcul**. Ce rapport permet de comparer les différents modèles du jeu de test par rapport aux caractéristiques quantitatives du *DG-ANDES*, car il s'agit d'une normalisation. Le rapport communication/calcul est un rapport entre le total estimé des temps de communication et le total estimé des temps de calcul. Dans le tableau qui suit, le suffixe "-a" correspond aux modèles avec un facteur *a* de communication, le suffixe "-b" correspond aux modèles avec un facteur *b* de communication et le suffixe "-c" correspond aux modèles avec un facteur *c* de communication :

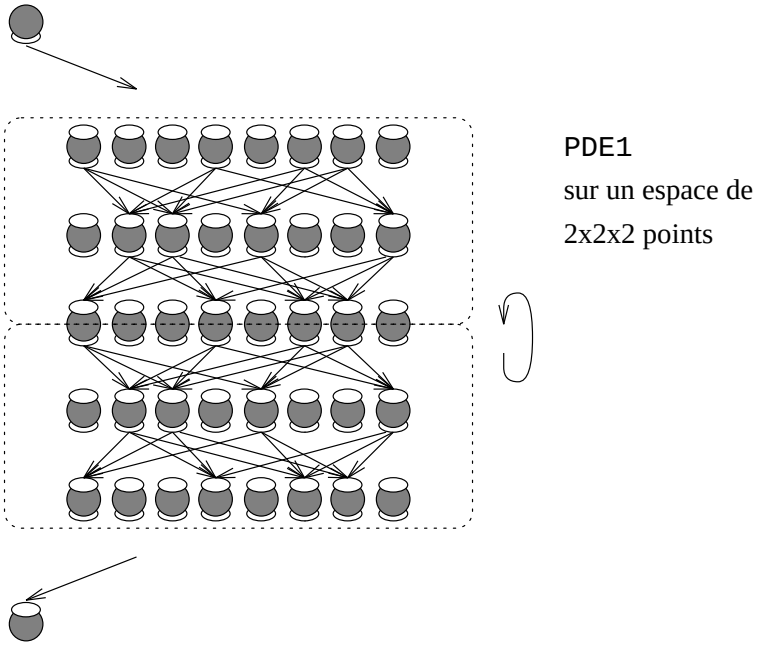



Figure 6.12 : La structure de PDE1.

| diam1-c | divcq-c | qcd02-c | divcq-b | diam3-c | prolg-c | divcq-a | msgas-c | ms3-c |
|---|---|---|---|---|---|---|---|---|
| 0,058 | 0,061 | 0,078 | 0,079 | 0,082 | 0,095 | 0,096 | 0,103 | 0,103 |
| bell2-c | gauss-c | diam1-b | diam3-b | bell2-b | stra2-c | fftr2-c | iter2-c | diam4-c |
| 0,107 | 0,123 | 0,123 | 0,177 | 0,227 | 0,250 | 0,254 | 0,313 | 0,341 |
| warsh-c | pde02-c | ms3-b | fftr2-b | qcd02-b | warsh-b | prolg-b | pde01-c | pde02-b |
| 0,344 | 0,369 | 0,393 | 0,402 | 0,487 | 0,524 | 0,596 | 0,613 | 0,620 |
| ms3-a | gauss-b | diam1-a | msgas-b | warsh-a | pde01-b | diam2-c | iter2-b | pde02-a |
| 0,756 | 0,770 | 0,771 | 0,779 | 0,884 | 1,030 | 1,093 | 1,102 | 1,121 |
| diam3-a | diam4-b | diam4-a | stra2-b | bell2-a | msgas-a | fftr2-a | pde01-a | iter2-a |
| 1,133 | 1,140 | 1,340 | 1,340 | 1,421 | 1,531 | 1,590 | 1,862 | 2,087 |
| qcd02-a | gauss-a | prolg-a | stra2-a | diam2-b | diam2-a | | | |
| 2,307 | 3,647 | 5,605 | 6,485 | 9,572 | 18,993 | | | |

La version *a* de DIAMOND2 présente presque 20 fois plus de temps de communication par rapport au temps de calcul. On espère que ce rapport diminue après l'utilisation des stratégies d'implémentation.

Les paramètres définis ci-dessus concernent le *DG-ANDES*. Cependant, les stratégies de placement prennent en compte un graphe non orienté. Nous avons besoin alors de faire un regroupement des nœuds de calcul afin d'augmenter le grain de calcul par rapport aux communications et de pouvoir utiliser les stratégies de placement disponibles. Le paragraphe suivant décrit les stratégies de regroupement et de placement employées.

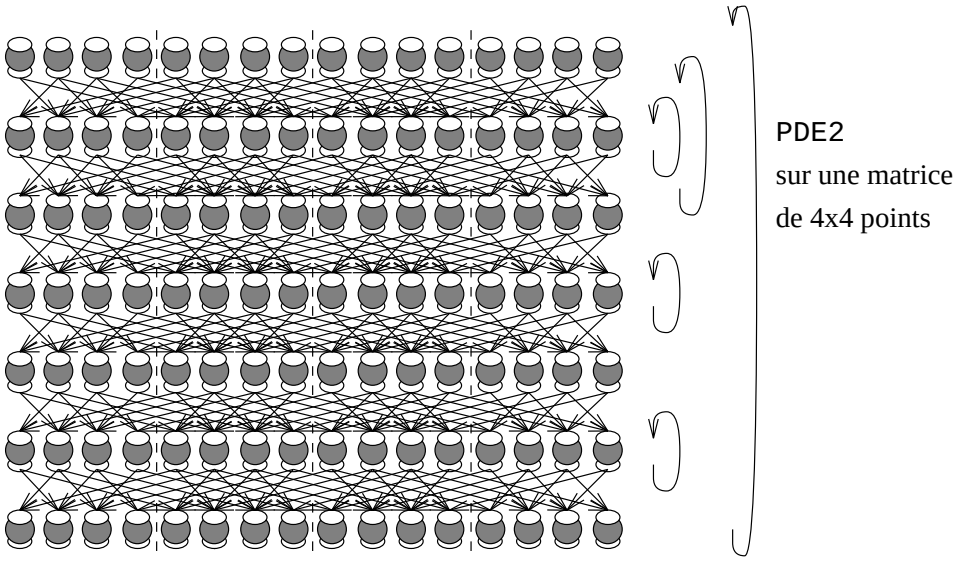



Figure 6.13 : La structure de PDE2.

### 6.2.5 Les stratégies d'implémentation : le regroupement

L'objectif de notre travail est la comparaison de stratégies de placement statique. Pour cela, il faut rendre compatible les entrées de telles stratégies et les *DG-ANDES* de notre jeu de test. Un regroupement doit alors être réalisé. Plusieurs stratégies de regroupement existent et l'outil *ANDES-Synth* peut être utilisé pour les comparer, mais ceci n'est pas notre objectif.

**Regroupement**. Pour regrouper les nœuds de calcul du *DG-ANDES*, nous avons utilisé deux stratégies : un regroupement **manuel** et un regroupement **automatique**. Pour le premier, nous avons mis un programmeur typique face au codage des algorithmes du jeu de test. En général, le programmeur essaye de trouver un compromis entre la performance de l'exécution du programme et la facilité de codage. Par exemple, un grand programme parallèle régulier est souvent codé comme un programme SPMD (un seul code objet dupliqué sur tous les processeurs), **même si ce codage ne conduit pas à l'exécution la plus performante**. Dans un *DG-ANDES*, le regroupement est spécifiée par les arcs du type CONTEXT. Deux nœuds de calcul liés par ce type de précédence s'exécutent forcément sur le même processeur. Aucun coût de communication n'est associé.

Si aucune précédence du type CONTEXT n'est spécifiée dans un modèle *ANDES*, l'outil considère qu'un regroupement automatique doit être réalisé. Le regroupement automatique prend en compte toujours les mêmes critères d'optimisation pour n'importe quel graphe. Nous avons utilisé pour cette deuxième approche l'algorithme DSC (*Dominant Sequence Algorithm*) développé par Gerasoulis & Yang [GY92]. L'objectif

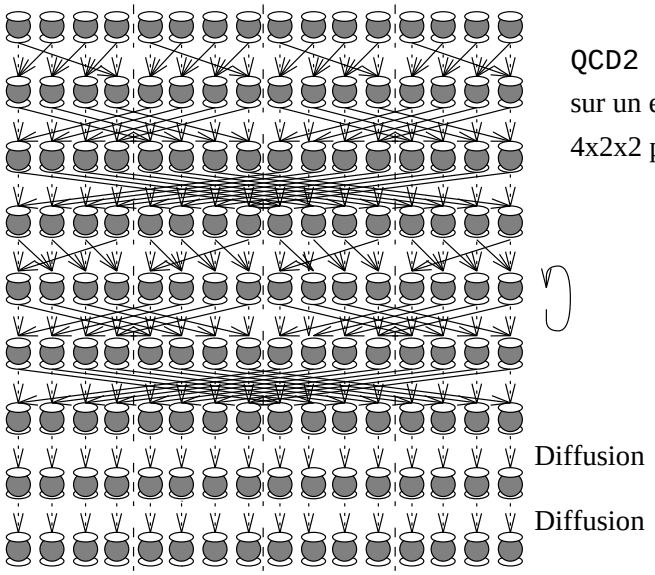
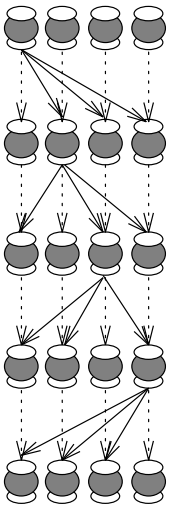



Figure 6.14 : La structure de QCD2.

de cet algorithme est la minimisation du temps d'exécution. Cet objectif est transformé en la minimisation de la séquence dominante du graphe analysé pendant l'itération courante de l'algorithme. Le calcul de la séquence dominante est réalisé de façon incrémentale pour éviter d'avoir un algorithme de grande complexité. La complexité de l'algorithme DSC est $(e + v) \log v$ ($v$ est le nombre de nœuds de calcul et $e$ est le nombre d'arcs du graphe). L'ordonnancement utilisé pour estimer le temps d'exécution est incrémental : un nœud de calcul du *DG-ANDES* est ordonnancé soit après le dernier nœud de calcul ordonnancé d'un groupe (du *DG-ANDES* ordonnancé dans l'itération précédente), soit comme le premier nœud de calcul ordonnancé dans un nouveau groupe. Rappelons que le modèle de machine pour un regroupement est un multiprocesseur, dont le graphe est une clique et tous les processeurs ont une même puissance de calcul.

**Paramètres du *DG-ANDES* groupé**. Revenons aux caractéristiques de notre jeu de test avec la connaissance du modèle de machine présenté ci-dessus et après avoir réalisé un regroupement manuel ou automatique en utilisant DSC. Nous regardons 3 caractéristiques des graphes groupés : le **degré moyen DM** (nombre moyen d'arêtes d'un groupe – caractéristique structurelle), le **rapport CC entre le total de temps estimé de communication et le total de temps estimé de calcul** (considérant les communications à distance 2) et la **régularité de communication RG** du graphe groupé. Le degré moyen représente un paramètre structurel important associé aux communications. Il est utilisé comme un critère décisif dans certaines stratégies de placement. En revanche, le rapport communication/calcul représente un critère quantitatif (voir le pa-



WARSHALL
taille du problème :
4 noeuds

Figure 6.15 : La structure de WARSHALL.

ragraphe 6.2.4). Ce rapport n'est pas le même que celui présenté pour le *DG-ANDES* : CC prend en compte l'annulation de communications réalisée par le regroupement.

Le paramètre régularité de calcul quantifie la régularité de la distribution de la communication entre deux groupes. Par exemple, si deux groupes $A$ et $B$ communiquent (par exemple, $A$ envoie des messages à $B$), nous voulons caractériser globalement ces communications : si elles ont lieu toutes au début de $A$ ou toutes à la fin ou si elles sont bien distribuées entre les deux groupes (Figure 6.16).

Un indice fidèle de cette régularité n'est pas facile à exprimer. Nous avons proposé une approximation donnée par la méthode suivante :

1. calcul d'un pourcentage normalisé ($R$) correspondant au temps global de communication ($TCOMM$ : la somme des coûts de communication de tous les arcs du *DG-ANDES* groupé) par rapport à la somme de tous les temps de communication et de calcul ($TCALC$ : la somme des coûts de calcul de tous les nœuds de calcul) :

$$R = \frac{\frac{TCOMM}{TCOMM}}{\frac{TCOMM}{TCOMM} + \frac{TCALC}{TCALC}} = \frac{1}{2}$$

2. ensuite, pour chaque groupe, tous les nœuds de calcul sont mis en séquence, sachant que ces nœuds seront exécutés de façon séquentielle par un seul processeur. Pour chaque groupe, ce procédé consiste à parcourir le sous-graphe par "étages" dans un groupe. Par exemple, supposons que $root$ soit le nœud de calcul racine d'un groupe (sans prédécesseur ou tous les prédécesseurs font partie d'un

| modèle | facteur *a* | facteur *b* | facteur *c* |
|---|---|---|---|
| BELLFORD2 | 10 | 1 | 0,1 |
| DIAMOND1 | 10 | 1 | 0,1 |
| DIAMOND2 | 2 | 1 | 0,1 |
| DIAMOND3 | 10 | 1 | 0,1 |
| DIAMOND4 | 1,2 | 1 | 0,2 |
| DIVCONQ | 1,5 | 1 | 0,5 |
| FFT2 | 5 | 1 | 0,5 |
| GAUSS | 5 | 1 | 0,1 |
| ITERATIVE2 | 2 | 1 | 0,2 |
| MS3 | 2 | 1 | 0,2 |
| MS_GAUSS | 2 | 1 | 0,1 |
| PDE1 | 2 | 1 | 0,5 |
| PDE2 | 2 | 1 | 0,5 |
| PROLOG | 10 | 1 | 0,1 |
| QCD2 | 5 | 1 | 0,1 |
| STRASSEN2 | 5 | 1 | 0,1 |
| WARSHALL | 2 | 1 | 0,5 |

Tableau 6.1 : Les facteurs de variation des coûts de communication.

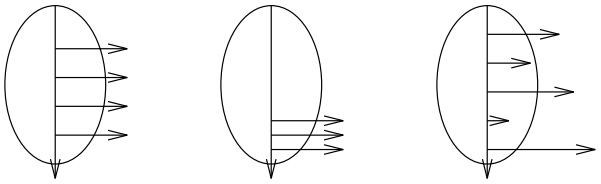
autre groupe). La mise en séquence consiste à considérer que tous les nœuds à distance 0 de $root$ (i.e., le $root$ lui-même) sont exécutés. Ensuite, on exécute tous les nœuds de calcul à distance 1 de $root$. On continue la procédure pour la distance 2, 3, …jusqu'au dernier nœud de calcul de ce groupe. On suppose que le sous-graphe des nœuds de calcul dans un groupe est connexe;

3. un groupe $g$ mis en séquence est une suite de durées de communication vers d'autres groupes ($tcomm_{ig}$) et de durées de calcul ($tcalc_{ig}$), $i = 1, \ldots, nt$; $g = 1, \ldots, ng$ ($nt$ est le nombre de nœuds de calcul de $g$ et $ng$ est le nombre de groupes). Nous nous intéressons au rapport $R_{ig}$ :

$$R_{ig} = \frac{\frac{tcomm_{ig}}{TCOMM}}{\frac{tcomm_{ig}}{TCOMM} + \frac{tcalc_{ig}}{TCALC}}, i = 1, \ldots, nt; g = 1, \ldots, ng$$

avec $tcomm_{1g} = 0$ toujours (il n y a pas de communication de sortie lorsque le groupe commence à s'exécuter – voir la Figure 6.17).

Si le groupe finit par une communication de sortie, alors $tcalc_{ntg} = 0$. La valeur de chaque $R_{ig}$ est comparée avec $R$ (le rapport global pour l'application). Le



régulier irrégulier irrégulier

* les tailles des flèches sont proportionnelles
  aux coûts de communication respectifs
* les distances entre les flèches sont proportionnelles
  aux durées d'exécution

Figure 6.16 : Exemples de régularité de communication.

coefficient de variation des $R_{ig}$ par rapport à $R$ correspond à notre paramètre "régularité de communication". La raison pour avoir choisi une relation entre un temps de communication de sortie et le temps du calcul qui **suit** cette communication est que nous pouvons, de cette façon, traiter le cas d'un groupe n'ayant qu'une communication de sortie à la fin. Si on considère ce cas très irrégulier (car une seule communication à la fin d'un groupe nous amène à un modèle de programme avec précédence), il y a deux communications (une nulle au début et une non nulle à la fin) et donc deux rapports $R_{ig}$ valant respectivement 0 et 1 qui ont un écart maximal par rapport à la moyenne 0,5. Si on considère la même formule pour le temps de communication de sortie et le temps du calcul qui **précède** cette communication, on n'observe pas ce même écart.

Afin de mieux comprendre le concept de la régularité, prenons un exemple d'extrême irrégularité donné par la Figure 6.18. Il y a $ng$ groupes, chacun avec un coût total de calcul $a$ et un coût de communication $b$. Pour chaque groupe, nous avons $R_{1g}$ et $R_{2g}$ :

$$R_{1g} = \frac{0}{0 + \frac{a}{TCALC}} = 0$$

$$R_{2g} = \frac{\frac{b}{TCOMM}}{\frac{b}{TCOMM} + 0} = 1$$

La variation $V$ est donnée par :

$$V = \frac{\sum_{g=1}^{ng}[(R_{1g} - R)^2 + (R_{2g} - R)^2]}{2ng - 1}$$

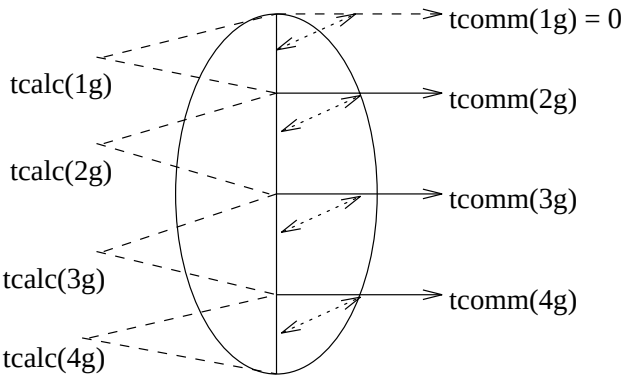



Figure 6.17 : Les rapports entre $tcomm$ et $tcalc$.

$$V = \frac{\sum_{g=1}^{ng}[(0-0,5)^2 + (1-0,5)^2]}{2ng-1}$$

$$V = \frac{\frac{2ng}{4}}{2ng-1}$$

L'écart-type $ET$ est donné par :

$$ET = \sqrt{V} = \frac{1}{2}\sqrt{\frac{2ng}{2ng-1}}$$

Si $ng \to \infty$, $ET = \frac{1}{2}$ et le coefficient de variation vaut $\frac{0,5}{0,5} = 1$. Par contre, si entre deux tâches il y a beaucoup de communications avec un $R_{ig} \sim \frac{1}{2}$, alors la variation $V$ sera petite. L'effet de $R_{1g} = 0$ (présent dans chaque groupe) est minimisé lorsqu'on a beaucoup de communications par groupe. Ainsi, la régularité idéale n'existe que pour un graphe avec un grand nombre de groupes et un grand nombre de communications (sortantes de façon régulière) par groupe.

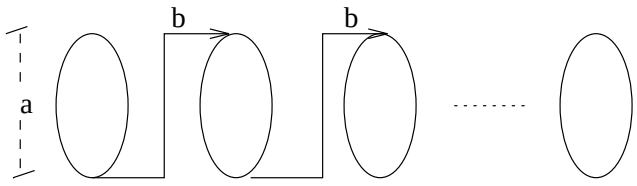



Figure 6.18 : Un graphe groupé irrégulier.

### 6.2.6 Paramètres du jeu de test - graphes groupés

**Degré moyen**. Pour le graphe groupé avec l'algorithme DSC, nous avons la caractérisation ci-dessous (unité des valeurs : arête par groupe). Il faut remarquer que l'algorithme DSC peut produire différentes topologies de graphes groupés à partir d'une même structure d'un *DG-ANDES*, si les rapports communication/calcul sont différents.

| gauss-a | diam1-a | diam1-b | diam1-c | diam2-a | diam2-b | iter2-a | iter2-b | iter2-c |
|---|---|---|---|---|---|---|---|---|
| 1,936 | 2,000 | 2,000 | 2,000 | 2,000 | 2,000 | 2,000 | 2,000 | 2,000 |
| prolg-a | diam2-c | prolg-b | prolg-c | divcq-a | divcq-b | divcq-c | ms3-a | ms3-b |
| 3,845 | 3,871 | 3,960 | 3,960 | 3,980 | 3,980 | 3,984 | 4,019 | 4,019 |
| ms3-c | stra2-a | stra2-b | stra2-c | diam4-a | diam4-b | diam4-c | diam3-c | diam3-b |
| 4,019 | 4,146 | 4,249 | 4,334 | 4,481 | 4,481 | 4,481 | 4,525 | 5,313 |
| diam3-a | pde01-a | pde01-b | pde01-c | pde02-a | pde02-b | pde02-c | bell2-a | bell2-b |
| 5,374 | 7,676 | 7,676 | 7,676 | 9,388 | 9,388 | 9,388 | 11,822 | 11,822 |
| bell2-c | fftr2-a | fftr2-b | fftr2-c | qcd02-a | qcd02-b | qcd02-c | msgas-a | msgas-b |
| 11,822 | 13,219 | 13,219 | 13,219 | 14,133 | 14,133 | 14,133 | 15,238 | 16,714 |
| msgas-c | warsh-a | warsh-b | warsh-c | gauss-b | gauss-c | | | |
| 16,714 | 17,609 | 17,609 | 17,609 | 30,065 | 30,065 | | | |

Pour les graphes groupés à la main (le regroupement est le même pour les trois rapports communication-calcul pour chaque modèle), nous avons :

| modèle | iter2 | divcq | diam1 | diam2 | ms3 | diam3 | diam4 | stra2 | prolg |
|---|---|---|---|---|---|---|---|---|---|
| DM | 1,992 | 1,998 | 2,000 | 2,000 | 3,000 | 3,604 | 3,604 | 3,930 | 3,988 |
| modèle | pde02 | pde01 | bell2 | qcd02 | fftr2 | msgas | gauss | warsh | |
| DM | 5,728 | 5,866 | 7,235 | 17,058 | 18,963 | 31,177 | 32,000 | 32,000 | |

**Rapport entre total de temps estimé de communication et total de temps estimé de calcul**. Pour les graphes groupés automatiquement :

| diam1-c | divcq-c | divcq-b | diam3-c | divcq-a | ms3-c | bell2-c | msgas-c | diam1-b |
|---|---|---|---|---|---|---|---|---|
| 0,021 | 0,028 | 0,037 | 0,040 | 0,047 | 0,055 | 0,059 | 0,059 | 0,061 |
| gauss-c | prolg-c | qcd02-c | diam3-b | stra2-c | fftr2-c | bell2-b | iter2-c | ms3-b |
| 0,063 | 0,063 | 0,072 | 0,117 | 0,144 | 0,147 | 0,171 | 0,206 | 0,208 |
| fftr2-b | diam4-b | warsh-c | pde02-c | diam1-a | gauss-b | prolg-b | ms3-a | warsh-b |
| 0,233 | 0,234 | 0,258 | 0,299 | 0,385 | 0,396 | 0,396 | 0,399 | 0,435 |
| diam2-b | qcd02-b | msgas-b | pde01-c | pde02-b | diam3-a | stra2-b | diam4-b | warsh-a |
| 0,442 | 0,448 | 0,451 | 0,494 | 0,502 | 0,745 | 0,764 | 0,782 | 0,788 |
| diam2-a | pde01-b | msgas-a | pde02-a | diam4-a | fftr2-a | iter2-b | diam2-c | bell2-a |
| 0,830 | 0,830 | 0,883 | 0,908 | 0,919 | 0,921 | 0,934 | 1,005 | 1,070 |
| pde01-a | gauss-a | iter2-a | qcd02-a | prolg-a | stra2-a | | | |
| 1,500 | 1,759 | 1,778 | 2,121 | 2,923 | 3,055 | | | |

Pour les graphes groupés manuellement :

| diam1-c | diam3-c | prolg-c | divcq-c | bell2-c | divcq-b | diam1-b | gauss-c | qcd02-c |
|---|---|---|---|---|---|---|---|---|
| 0,021 | 0,040 | 0,040 | 0,042 | 0,058 | 0,061 | 0,062 | 0,063 | 0,073 |
| divcq-a | msgas-c | ms3-c | diam3-b | stra2-c | fftr2-c | bell2-b | diam2-c | iter2-c |
| 0,080 | 0,091 | 0,092 | 0,120 | 0,131 | 0,140 | 0,169 | 0,171 | 0,208 |
| prolg-b | fftr2-b | diam4-c | warsh-c | pde02-c | ms3-b | diam1-a | gauss-b | warsh-b |
| 0,210 | 0,221 | 0,227 | 0,261 | 0,294 | 0,382 | 0,385 | 0,397 | 0,441 |
| qcd02-b | pde02-b | pde01-c | ms3-a | msgas-b | diam4-b | diam3-a | warsh-a | diam2-b |
| 0,457 | 0,494 | 0,504 | 0,744 | 0,756 | 0,759 | 0,775 | 0,800 | 0,808 |
| pde01-b | stra2-b | fftr2-a | diam4-a | pde02-a | iter2-b | bell2-a | msgas-a | diam2-a |
| 0,846 | 0,847 | 0,874 | 0,891 | 0,894 | 0,971 | 1,058 | 1,494 | 1,515 |
| pde01-a | gauss-a | prolg-a | iter2-a | qcd02-a | stra2-a | | | |
| 1,530 | 1,880 | 1,902 | 1,926 | 2,166 | 4,011 | | | |

**Régularité de communication**. Pour les graphes groupés automatiquement :

| diam1-a | diam1-b | diam1-c | qcd02-a | qcd02-b | qcd02-c | diam2-c | ms3-c | ms3-b |
|---|---|---|---|---|---|---|---|---|
| 0,252 | 0,252 | 0,252 | 0,258 | 0,258 | 0,258 | 0,258 | 0,262 | 0,275 |
| ms3-a | diam2-a | diam2-b | msgas-c | msgas-b | msgas-a | diam3-c | diam3-a | diam3-b |
| 0,277 | 0,292 | 0,292 | 0,338 | 0,345 | 0,345 | 0,445 | 0,446 | 0,446 |
| pde01-a | pde01-b | pde01-c | pde02-a | pde02-b | pde02-c | fftr2-a | fftr2-b | fftr2-c |
| 0,480 | 0,480 | 0,480 | 0,510 | 0,510 | 0,510 | 0,513 | 0,513 | 0,513 |
| bell2-a | bell2-b | bell2-c | diam4-a | diam4-b | diam4-c | prolg-a | stra2-a | prolg-b |
| 0,519 | 0,519 | 0,519 | 0,577 | 0,577 | 0,577 | 0,774 | 0,803 | 0,817 |
| prolg-c | divcq-a | divcq-b | divcq-c | stra2-b | stra2-c | warsh-c | warsh-b | warsh-a |
| 0,817 | 0,830 | 0,831 | 0,834 | 0,898 | 0,933 | 0,956 | 0,956 | 0,956 |
| gauss-a | iter2-a | iter2-b | iter2-c | gauss-b | gauss-c | | | |
| 0,985 | 1,000 | 1,000 | 1,000 | 1,005 | 1,005 | | | |

Pour les graphes groupés manuellement :

| ms3-a | ms3-b | ms3-c | msgas-a | msgas-b | msgas-c | diam1-a | diam1-b | diam1-c |
|---|---|---|---|---|---|---|---|---|
| 0,125 | 0,125 | 0,125 | 0,190 | 0,190 | 0,190 | 0,252 | 0,252 | 0,252 |
| diam2-a | diam2-b | diam2-c | pde02-a | pde02-b | pde02-c | fftr2-a | fftr2-b | fftr2-c |
| 0,252 | 0,252 | 0,252 | 0,414 | 0,414 | 0,414 | 0,428 | 0,428 | 0,428 |
| diam3-c | diam3-b | diam3-a | diam4-a | diam4-b | diam4-c | bell2-a | bell2-b | bell2-c |
| 0,445 | 0,445 | 0,446 | 0,452 | 0,452 | 0,452 | 0,454 | 0,454 | 0,454 |
| pde01-a | pde01-b | pde01-c | qcd02-a | qcd02-b | qcd02-c | prolg-c | warsh-c | warsh-b |
| 0,579 | 0,579 | 0,579 | 0,641 | 0,641 | 0,641 | 0,849 | 0,869 | 0,869 |
| warsh-a | divcq-c | divcq-b | divcq-a | stra2-a | stra2-b | stra2-c | prolg-b | prolg-a |
| 0,869 | 0,900 | 0,901 | 0,902 | 0,918 | 0,918 | 0,921 | 0,941 | 0,979 |
| iter2-a | iter2-b | iter2-c | gauss-a | gauss-b | gauss-c | | | |
| 1,000 | 1,000 | 1,000 | 1,006 | 1,006 | 1,006 | | | |

Les paramètres du jeu de test (les *DG-ANDES* et les graphes groupés) présentés dans ce chapitre sous la forme de tableaux peuvent être résumés comme suit :

- **Parallélisme virtuel** : `MS3`, `QCD2`, `GAUSS`, `DIAMOND1` et `DIAMOND2` ont un nombre moyen faible de nœuds de calcul pouvant s'exécuter en parallèle (moins de 16, le nombre de processeurs de la machine modélisée). `DIAMOND4`,

MS_GAUSS, BELLFORD2, WARSHALL et DIAMOND3 ont un nombre moyen entre 20 et 34. PROLOG, DIVCONQ et STRASSEN2 forment un autre groupe avec un parallélisme virtuel entre 77 et 92 (environ 5 à 6 fois plus grand que le nombre de processeurs de la machine modélisée). ITERATIVE2, PDE2, PDE1 et FFT2 (des algorithmes typiquement itératifs) ont beaucoup (> 200) de nœuds qui peuvent être exécutés en parallèle;

- **Degré moyen du graphe groupé** : les graphes groupés à la main ont, en général, un degré moyen plus grand que les graphes correspondants groupés par DSC. ITERATIVE2, DIVCONQ, DIAMOND1, DIAMOND2, MS3, DIAMOND3, DIAMOND4, STRASSEN2 et PROLOG ont peu de voisins (< 4,1) pour les deux jeu de tests. Au contraire, les graphes groupés à la main de PDE1, PDE2 et BELLFORD2 ont moins de voisins que leurs correspondants dans le regroupement automatique. FFT2, QCD2, MS_GAUSS, GAUSS et WARSHALL ont beaucoup (> 10) voisins, sauf la version de GAUSS avec le plus grand rapport communication/calcul. Ce modèle a le moins de voisins (1,936) pour un regroupement automatique.

- **Rapport entre le temps de communication et le temps de calcul pour les graphes groupés** : ces rapports sont régulièrement distribués entre 0 et 1 pour les deux jeux de test. Les modèles qui ont les rapports les plus grands (> 1) sont les versions '-a' de PDE1, de GAUSS, de ITERATIVE2, de QCD2, de PROLOG et de STRASSEN2;

- **Régularité de communication** : la régularité de communication correspond à un coefficient de variation. Plus bas sont ces coefficients, plus régulières sont les communications. On trouve DIAMOND1, DIAMOND2, DIAMOND3, DIAMOND4, MS3 et MS_GAUSS les graphes les plus réguliers par rapport à la communication. ITERATIVE2 est très irrégulier à cause du choix aléatoire des successeurs à chaque étage du graphe. L'irrégularité de GAUSS vient du regroupement non équilibré par colonne par rapport aux communications.

Nous considérons le modèle ITERATIVE2 très particulier par rapport aux autres : fort parallélisme virtuel, un nombre faible de voisins par groupe, un grand rapport communication/calcul et irrégulier en communication. Les modèles PDE1, PDE2, FFT2, QCD2, PROLOG et STRASSEN2 ont des caractéristiques particulières pour certains critères.

### 6.2.7 Les stratégies d'implémentation : le placement

Le *DG-ANDES* étant groupé, il est possible d'appliquer des stratégies de placement statique. Six algorithmes ont été employés dans nos expériences : 4 algorithmes gloutons (modulo, lpt, quanti et struc_quanti) et 2 algorithmes itératifs (qsiman et qtabu) [Bou94].

`modulo`. Il s'agit d'un algorithme glouton qui consiste à placer le $i$-ème groupe sur le *i modulo P* processeur ($P$ : nombre de processeurs).

Avantages et désavantages :

- $\oplus$ cet algorithme est très rapide ($O(n)$);
- $\oplus$ si la variation des coûts de calcul des groupes est petite, cet algorithme établit un bon équilibrage de charge;
- $\oplus$ si les identificateurs (les numéros) des groupes ayant le même poids de calcul sont consécutifs, cet algorithme établit un bon équilibrage de charge;
- $\ominus$ cet algorithme ne prend ni en compte les coûts de calcul, ni les coûts de communication (pas de fonction de coût à évaluer). Cela est particulièrement mauvais dans les cas où :
  - les calculs prédominent et il y a une grande variation entre ses coûts;
  - les communications prédominent.

`lpt`. Cet algorithme est un algorithme de liste qui essaye d'équilibrer les coûts de calcul des groupes entre les différents processeurs. Les groupes sont tout d'abord triés par ordre décroissant de coût de calcul (le plus grand coût de calcul d'abord – en anglais LPTF *Largest Processing Time First*). Ensuite le premier groupe est placé sur un processeur quelconque et le $i$-ème groupe sur le processeur le moins chargé au niveau calcul.

Avantages et désavantages :

- $\oplus$ cet algorithme est très rapide ($O(n)$);
- $\oplus$ cet algorithme établit un bon équilibrage de charge de calcul, qui est au pire à $\frac{4}{3}$ de l'équilibrage optimal, pour un modèle de machine sans coût de communication;
- $\ominus$ cet algorithme ne prend pas en compte les coûts de communication.

`quanti`. Cet algorithme est aussi connu comme "le plus grand coût total d'abord" (LGCF – en anglais *Largest Global Cost First*). `quanti` essaye d'équilibrer un **coût global** (qui considère le calcul et la communication) entre les différents processeurs. Les groupes sont tout d'abord triés par ordre décroissant de coût global. Le coût global d'un groupe correspond à la somme de son temps de calcul et de son temps de communication. Le temps de calcul d'un groupe est la somme de tous les temps de calcul des nœuds du *DG-ANDES* qui font partie de ce groupe. Le temps de communication d'un groupe est la somme de tous les temps de communication des nœuds du *DG-ANDES* qui font partie de ce groupe (les communications intra-groupe ont une durée nulle).

Ensuite le premier groupe est placé sur un processeur quelconque et le $i$-ème groupe sur le processeur moins chargé. La charge d'un processeur est exprimée par la somme des coûts globaux de tous les groupes placés sur ce processeur.

Avantages et désavantages :

- $\oplus$ cet algorithme est très rapide ($O(n)$);
- $\oplus$ cet algorithme tente d'établir un bon équilibrage de charge de calcul et de communication;
- $\ominus$ aucun effort pour effectuer des communications intra-processeur n'est fait, c'est-à-dire que la topologie du graphe de groupes n'est pas utilisée dans les choix réalisés par le placement.

`struc_quanti`. Cet algorithme est similaire à `quanti`, mais le critère de tri des groupes est le nombre de liens de communications des groupes (le degré). Si deux groupes ont le même nombre de liens, alors ils sont triés par ordre décroissant de coût global. Le $i$-ème groupe est placé sur le processeur le moins chargé globalement.

`qsiman`. Cet algorithme est une implémentation de la méthode d'optimisation connue sous le nom de "recuit simulé" (en anglais *simulated annealing* [BA92]). L'idée de cette méthode provient de la physique : 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. Dans le cadre du placement, ce principe est employé de la façon suivante : partant d'un placement initial complet, on va chercher à l'améliorer par un critère local tel que l'échange de paires de groupes. 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 afin de sortir d'un minimum local de "l'énergie" (fonction à optimiser). Pour `qsiman`, la solution de départ est donnée par l'algorithme `quanti`.

Le schéma de la méthode du recuit simulé est donné dans la Figure 6.19.

Avantages et désavantages :

- $\oplus$ les placements obtenus sont en moyenne assez bons et il existe des preuves de convergence vers une solution optimale dans le cas où la température décroît très lentement;
- $\ominus$ temps de calcul élevé, devenant vite disqualifiant pour un nombre important de groupes;
- $\ominus$ son comportement est imprévisible à cause de modifications aléatoires du placement;

⊖ cet algorithme dépend d'un certain nombre de paramètres (e.g., la température de départ et d'arrivée, la vitesse de refroidissement) qui sont difficiles à ajuster et dépendent en fait du placement à effectuer.

`qtabu`. Au cœur de la recherche tabou [Glo90], 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). Les relations de voisinage permettent de se promener dans l'espace de solutions. Cette promenade devrait permettre le passage par un optimum global ou au moins le passage par des minima locaux intéressants. A chaque itération, l'ensemble des candidats que l'on peut atteindre par les relations de voisinage est examiné. Plus les relations de voisinage sont étendues, plus on aura de chance de trouver une bonne solution, au prix d'augmenter aussi le coût d'évaluation de chaque itération. Une mémoire est utilisée afin de retenir la meilleure solution trouvée. Afin d'éviter de revenir sur une solution déjà explorée, une mémoire limitée des solutions parcourues est établie : la **liste tabou**. Le **critère d'aspiration** doit permettre de revenir sur une solution qui n'est pas admise par la liste tabou. Plusieurs **critères d'arrêt** sont possibles : par exemple, si une solution satisfaisante a été rencontrée ou si aucune amélioration de la meilleure solution rencontrée n'a été trouvée depuis longtemps. Finalement, les fonctions d'identification et de diversification permettent de s'attarder ou de s'éloigner de voisinages qui semblent ou ne semblent pas intéressants.

Dans `qtabu`, la solution de départ est donnée par l'utilisation de `quanti`. Deux types de voisinage sont utilisés : le transfert d'un groupe du processeur le plus chargé globalement et l'échange d'un groupe situé sur le processeur le plus chargé avec un autre groupe communiquant avec celui-ci et situé sur un autre processeur. Chaque élément de la liste tabou contient une description du déplacement d'un groupe vers un processeur. La taille de la liste tabou a été fixée empiriquement en relation avec le nombre de groupes [Bou94]. Le critère d'aspiration est l'amélioration de la meilleure valeur rencontrée de la fonction de coût.

Le schéma de la méthode de la recherche tabou est donné dans la Figure 6.20.

| Nom de variable | Type | But |
|---|---|---|
| heat | réel | Température du recuit |
| minimum | réel | Température finale |
| local_equilibrium | booléen | L'équilibre local est-il atteint?? |
| move | {(task,dest)} | Transformations 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 |
| 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_sol);
            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;
      };
```

Figure 6.19 : Le recuit simulé [Bou94].

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

```
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);
      };
```

Figure 6.20 : La recherche tabou [Bou94].

Avantages et désavantages :

⊕ pour certains types de problème (e.g., *job-shop*), les algorithmes tabou donnent les meilleurs résultats;

⊖ ces algorithmes sont en général plus difficiles à mettre en œuvre que les recuits simulés.

**Fonctions de coût**. Diverses fonctions de coût ont été prises en compte pour les stratégies décrites ci-dessus. La fonction de coût de base $f_1$ tente de trouver un compromis entre la minimisation des communications et l'équilibrage de charge de calcul :

$$f_1 = \max_{p_k \in P} \sum_{t_i | alloc(t_i)=p_k} [calc(t_i) + \sum_{t_j | alloc(t_j) \neq alloc(t_i)} comm(t_i; t_j)]$$

Les coûts sont des moyennes et en unité de temps. S'il y a un recouvrement du calcul et de communication (la communication peut être exécutée en parallèle avec le calcul), l'expression ci-dessus devient :

$$f_2 = \max_{p_k \in P} [\max_{t_i | alloc(t_i)=p_k} [calc(t_i), \sum_{t_j | alloc(t_j) \neq alloc(t_i)} comm(t_i; t_j)]]$$

Afin de tenir compte des distances inter-processeurs de la machine modélisée, une fonction de coût basée sur la table de routage a été définie :

$$f_3 = \max_{p_k \in P} \sum_{t_i | alloc(t_i)=p_k} [calc(t_i) + \sum_{t_j | alloc(t_j) \neq alloc(t_i)} \delta_{alloc(t_i), alloc(t_j)} comm(t_i; t_j)]$$

où $\delta_{a,b}$ représente le nombre de sauts à effectuer entre le processeur $a$ et le processeur $b$. Finalement, une fonction de coût précise $f_4$ a été définie à partir de mesures sur la machine cible (un tore 4x4 plus 2 processeurs auxiliaires) selon la même idée que $f_3$. Les modèles de communication considérés sont présentés dans le paragraphe 6.2.2. En conclusion, nous avons essayé 4 fonctions de coût différentes (sans recouvrement - $f_1$, avec recouvrement - $f_2$, une fonction multiplicative par la distance entre les processeurs - $f_3$ et la fonction de coût précise du tore - $f_4$). Toutes les fonctions de coût ne prennent pas en compte l'effet dû à l'encombrement du réseau. Nous appellerons $f_1$, $f_2$, $f_3$ et $f_4$ respectivement ADD, MAX, ROUT et TOR.

Avant de présenter les résultats de nos expériences, nous donnons quelques paramètres expérimentaux auxiliaires utilisés.

### 6.2.8 Les paramètres auxiliaires des expériences

Quelques paramètres auxiliaires ont été définis pour nos expériences :

- **taille de l'échantillon pour chaque mesure**. Etant donnée une charge synthétique dont le placement a été choisi par une stratégie spécifique et s'exécutant avec un degré de multiprogrammation donné, plusieurs mesures ont été faites. En effet, nous avons constaté que la variation entre plusieurs mesures pour chaque modèle (et pour une stratégie de placement) est faible, mais elle existe. Une variation des temps mesurés pour un modèle donné et pour différentes stratégies existe aussi. Il faut que la taille de l'échantillon soit assez grande pour que l'on puisse discriminer les temps mesurés donnés par les différentes stratégies. Afin d'estimer la variation, nous avons réalisé une expérience. Nous avons exécuté 11 fois la charge synthétique correspondante à chaque modèle du jeu de test. Nous avons pris les modèles avec le facteur *a* de communication. Avec cette condition, on espère une variation grande du temps d'exécution à cause de la synchronisation entre les nœuds de calcul. Les coefficients de variation obtenus sont plus petits que 2,6% (voir le Tableau 6.2). Si on considère $\frac{\sigma}{\sqrt{n}}$ une approximation de l'erreur standard pour la moyenne ($\sigma$ correspond à l'écart et $n$ à la taille de l'échantillon) [Jai91], un $n = 100$ est suffisant pour donner un intervalle de 0,52% ($\frac{2\times 0,026}{\sqrt{100}} = 0,0052$). Ainsi, si la différence entre les temps moyens donnés par les différentes stratégies est de $2 \times 0,52 = 1,04$ %, on ne peut rien conclure;

- **temps d'exécution maximum pour les stratégies de placement**. Nous avons fixé 3600 secondes la durée maximale d'exécution des algorithmes de placement statique (une limite qui est suffisamment au-delà de la durée d'exécution des algorithmes parallèles modélisés par le jeu de test).

## 6.3 Résultats et analyse

Tout d'abord nous avons réalisé deux séries d'expériences avec le jeu de test donné : une série avec le regroupement à la main et autre série avec le regroupement automatique. Pour chaque modèle de programme, nous disposons de trois rapports différents de communication/calcul. Nous avons utilisé les stratégies de placement décrites dans le paragraphe 6.2.5. Dans ce travail, nous considérons le temps total d'exécution comme indice de performance. Pour analyser les mesures obtenues, nous utilisons la méthode de l'analyse en composantes principales décrite dans le paragraphe suivant.

### 6.3.1 La démarche statistique

La démarche statistique adoptée dans notre travail est liée à la **statistique exploratoire** [Sap90]. Son objectif est de synthétiser, de résumer et de structurer l'information contenue dans les données afin de **mettre en évidence des propriétés de l'échantillon** et **de suggérer des hypothèses**. Une étude plus approfondie peut être réalisée à partir de cette phase exploratoire : on entre alors dans le domaine de la **statistique inféren-**

| modèle | coefficient de variation pour 11 exécutions |
|---|---|
| BELLFORD2 | 0,0212758 |
| DIAMOND1 | 0,0248429 |
| DIAMOND2 | 0,0156542 |
| DIAMOND3 | 0,0139188 |
| DIAMOND4 | 0,0188087 |
| DIVCONQ | 0,0262802 |
| FFT2 | 0,0109647 |
| GAUSS | 0,0000936 |
| ITERATIVE2 | 0,0000916 |
| MS3 | 0,0005024 |
| MS_GAUSS | 0,0089903 |
| PDE1 | 0,0098611 |
| PDE2 | 0,0071310 |
| PROLOG | 0,0045877 |
| QCD2 | 0,0110189 |
| STRASSEN2 | 0,0229240 |
| WARSHALL | 0,0053778 |

Tableau 6.2 : Les coefficients de variation des exécutions des charges synthétiques.

**tielle**, dont le but est d'étendre les propriétés constatées sur l'échantillon à la population entière et de valider des hypothèses formulées. Cependant la qualité des résultats obtenues dans la statistique inférentielle est dépendante de la taille de l'échantillon qui doit être assez grande. Nous traitons une population à faible nombre d'individus ce qui nous empêche d'aller plus loin dans notre analyse.

Les principales méthodes de l'analyse de données pour la statistique exploratoire sont :

- les **méthodes de classification** qui visent à réduire la taille de l'ensemble des individus en formant des groupes homogènes;
- les **méthodes factorielles** qui cherchent à réduire le nombre de variables (les caractéristiques des individus de la population) en les résumant par un petit nombre de composantes synthétiques. L'**analyse en composantes principales** est la méthode factorielle typique pour une population caractérisée par des variables numériques.

Nous avons choisi l'analyse en composantes principales pour analyser les mesures obtenues avec l'exécution des charges synthétiques. Les méthodes de classification

sont, en général, très gourmandes en calcul et en mémoire. Le manque d'un bon logiciel pour cette méthode a joué aussi dans notre choix.

### 6.3.2 L'analyse en composantes principales

Le principe de l'analyse en composantes principales (l'ACP) est d'obtenir une représentation approchée de l'échantillon de $n$ individus (le nuage des individus) 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 cœur de cette méthode est un **changement de base** dans l'espace des individus. La base initiale est composée de $v$ variables originales. On réalise des **combinaisons linéaires** de ces variables pour former la nouvelle base (orthogonale) : le premier vecteur de la base (le premier **facteur** ou **axe** principal) est celui qui donne la direction de la plus grande dispersion du nuage, le second est **orthogonal** au premier et donne la direction de la seconde plus grande dispersion du nuage, et ainsi de suite.

L'ACP construit de nouvelles variables artificielles, et des représentations graphiques permettant de visualiser les relations entre les variables, ainsi que l'existence éventuelle de groupes d'individus et de groupes de variables.

**L'interprétation graphique de l'ACP [Ber94]**. L'ACP réalisée dans ce travail traite des variables quantitatives **centrées** (sa moyenne est ramenée à 0) et **réduites** (la variable centrée est divisée par son écart-type). Une variable réduite n'a plus d'unité, ce qui permet de comparer des variables de différents natures (e.g., microseconds, octets). La représentation graphique est donnée par les projections du nuage sur les plans déterminés par les axes principaux. Par exemple, le "plan 1-2" est le graphique ayant les projections du nuage sur le plan engendré par les axes 1 et 2. Il faut interpréter l'ACP avec la connaissance de ce que toute projection déforme. Les variables sont représentées dans l'espace par des vecteurs, dont la norme correspond à leur écart-type (1 pour les variables réduites). Le cosinus de l'angle entre deux vecteurs est égal au coefficient de corrélation linéaire entre les deux variables qu'ils représentent. Les variables sont alors tracées sur un **cercle de corrélation** (de rayon 1 et le centre à l'origine des axes). Plus une variable est proche du cercle, mieux elle est représentée, et plus il sera possible de donner une interprétation de cette variable dans le plan considéré. Si deux variables sont bien représentées, l'angle entre les deux vecteurs correspondants donnent le degré de corrélation linéaire entre les variables. Du côté de la représentation des individus, il est nécessaire de savoir si un individu est proche (respectivement éloigné) du plan de projection, afin de déterminer s'il est bien représenté (respectivement mal représenté) sur le graphique. Le critère utilisé est $\cos^2 \theta_i$ (si le cosinus vaut 1, la représentation est parfaite). $\theta_i$ est l'angle entre l'individu $i$ et le plan de projection. Pour l'analyse d'un plan donné, on ne peut parler que des individus qui y sont bien représentés. Il nous reste à présenter une méthode pour déterminer le nombre d'axes (ou de facteurs) à retenir. Malheureusement, les critères utilisés pour cela sont empiriques. Lorsqu'on

réalise une ACP, les axes principaux sont en fait les vecteurs propres $(v_k)_{1 \leq k \leq n}$ de la matrice $MV$ ($M$ est une matrice associée à la métrique utilisée dans l'ACP et $V$ est la matrice de variances et covariances empiriques). Ces vecteurs propres sont rangés dans l'ordre décroissant des valeurs propres $(\lambda_k)_{1 \leq k \leq n}$ (un exemple est présenté dans la Figure 6.21).

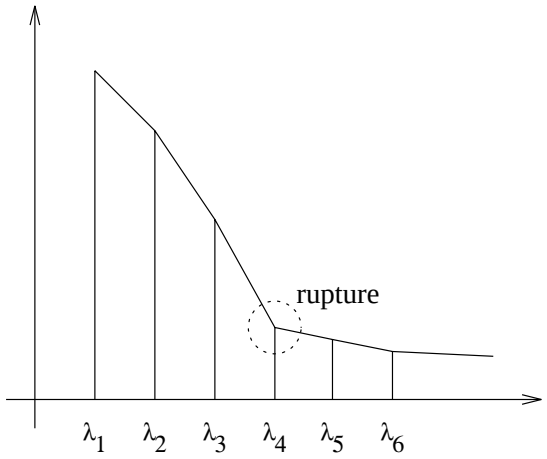



Figure 6.21 : Le diagramme des valeurs propres [Sap90].

L'ACP est satisfaisante si les premières valeurs propres sont de même ordre de grandeur, et si on l'observe une rupture de pente après 3 ou 4 axes. Selon le critère de Kaiser, on retient les composantes principales correspondant à des valeurs propres supérieures à 1 : on s'intéresse seulement aux composantes de variance supérieure à celle des variables initiales.

Une dernière remarque sur l'ACP est l'aspect qui concerne les éliminations successives de variables. Si le nombre de variables est important, plusieurs problèmes peuvent arriver :

- il y peut avoir la redondance de l'information exprimée par les variables, car quelques unes de ces variables peuvent être très corrélées;
- la représentation graphique des variables est surchargée;
- l'éboulis de valeurs propres n'est pas satisfaisant : en général il n'y a plus de rupture de pente à partir de la deuxième valeur propre.

Il est alors souhaitable d'éliminer des variables. Comme pour le choix des axes principaux, cette élimination est faite de façon empirique et parfois contestable. C'est la bonne connaissance de la nature des variables qui joue dans ce procédé empirique. Afin d'éliminer des variables, on utilise principalement les graphes des variables et leurs

corrélations linéaires. Les variables les plus redondantes sont éliminées. A chaque élimination, une ACP sur les variables restantes est refaite. A la fin, pour chaque variable éliminée, on doit alors vérifier qu'elle peut être modélisée à partir de celles qui restent.

Dans le cadre de nos travaux, les analyses en composantes principales ont été réalisées en utilisant le logiciel `xlispstat` sur Unix et le programme `ACPdialog.lsp`, développé par Christian Lavergne à l'ENSIMAG/INPG[2]. Les logiciels `xlispstat` et `ACPdialog.lsp` sont distribués gratuitement.

### 6.3.3 Les résultats

Des exécutions des jeux de test ont été faites. Le résultat d'une exécution est le temps parallèle d'exécution. On examine aussi l'accélération obtenue qui est calculée comme le rapport entre la somme des coûts de calcul d'un graphe groupé et le temps d'exécution parallèle mesuré. Ce rapport est connu en anglais sous le nom de *speedup*. A partir des résultats obtenus, plusieurs études ont été réalisées. Tout d'abord nous étudions la qualité des fonctions de coûts. Ensuite, nous vérifions, par le moyen de la matrice de corrélations linéaires, les dépendances entre les paramètres (quantitatifs et structurels) des modèles des jeux de test utilisés. Nous réalisons alors une ACP sur la matrice d'accélérations afin de déterminer quelles stratégies ont un bon comportement et sous quelles conditions.

**Les fonctions de coût**. La fonction de coût employée par les stratégies de placement tentent d'estimer le temps d'exécution du programme associé au graphe de communication valué. La fonction de coût parfaite est celle qui prévoit avec exactitude ce temps d'exécution. Sur un graphe [fonction de coût versus temps d'exécution parallèle], le comportement de la fonction de coût idéale est représenté par une droite $y = x$. Une bonne fonction de coût est celle dont le comportement dans ce plan [fonction de coût versus temps d'exécution parallèle] est représenté par une droite $y = \alpha + x$ (c'est-à-dire, parallèle à $y = x$), le $\alpha$ étant une différence expliquée par la surcharge due au système d'exploitation et à l'outil *ANDES-Synth*. Pour comparer les différentes fonctions de coût employées par les stratégies de placement, nous avons réalisé une régression linéaire des points [valeur de la fonction de coût,temps mesuré] afin de vérifier les pentes de chaque modèle obtenu. Les qualités des régressions réalisées sont moyennes, mais le modèle nous laissent voir les tendances de chaque droite. Le Tableau 6.3 est un sommaire des valeurs obtenues. Les fonctions de coût MAX, ADD, ROUT et TOR sont explicitées à la fin du paragraphe 6.2.5.

Nous remarquons que TOR est la meilleure fonction de coût pour les deux cas. MAX et ROUT ne représentent pas le comportement de la machine modélisée. D'abord, le recouvrement du calcul avec la communication n'est pas parfait à cause de la communication semi-bloquante (le rendez-vous partiel – le calcul n'attend que la réception du premier paquet) et principalement à cause du routage réalisé par logiciel :

[2] Christian Lavergne - Ensimag - BP 53X - 38051, Grenoble CEDEX - France

| | ADD | MAX | TOR | ROUT | Regroupement |
|---|---|---|---|---|---|
| pente | 1,213 | 1,620 | 1,185 | 1,636 | automatique |
| corrélation linéaire | 0,923 | 0,894 | 0,907 | 0,947 | automatique |
| pente | 1,370 | 2,147 | 1,302 | 1,664 | manuel |
| corrélation linéaire | 0,734 | 0,796 | 0,794 | 0,771 | manuel |

Tableau 6.3 : Les pentes et les coefficients de corrélation linéaires entre les valeurs des fonctions de coût et les temps mesurés.

le temps dépensé dans l'acheminement de paquets est compté dans le temps d'exécution parallèle. Les différences entre ADD et TOR ne sont pas significatives, car le routage en pipeline de la machine modélisée cache la distance entre les multiprocesseurs. En d'autres termes, pour la communication d'un grand message, la distance entre le processeurs a peu d'influence sur le temps de communication de ce message.

Une autre conclusion tirée du Tableau 6.3 concerne la qualité du regroupement manuel. On remarque que le regroupement manuel est issu d'un compromis entre la facilité de codage et la performance de l'exécution parallèle. Il n'y a pas un critère unique et formel comme dans le regroupement réalisé par l'algorithme DSC (la minimisation de la séquence dominante et, par conséquent, du temps d'exécution parallèle qui est le même but des fonctions de coût utilisées par les stratégies de placement). Les valeurs de pentes sont plus grandes dans le regroupement manuel que dans le regroupement automatique. Donc le regroupement manuel prépare moins bien le placement.

Une remarque importante : même pour TOR, la pente est supérieure à 1. Ceci signifie que plus la charge imposée par le programme est forte, moins la valeur de la fonction de coût est proche du coût réel d'exécution. En effet, la charge du réseau n'est pas modélisée dans les fonctions de coût employées ici, qui donnent donc des valeurs optimistes. Par conséquent, la différence entre la valeur de la fonction de coût et le temps mesuré est plus grande.

Afin de faire une analyse plus détaillée du comportement de la fonction de coût TOR, nous avons comparé les variations des temps mesurés en fonction des variations des valeurs de la fonction de coût. Etant donné que les différentes stratégies de placement donnent des différentes valeurs de fonction de coût et des différentes durées d'exécution, nous avons comparé deux à deux toutes les variations possibles (e.g., en appliquant `lpt` et `quanti`, en appliquant `lpt` et `qtabu`, etc…). Nous vérifions que dans 48,3% des cas, les variations vont dans le même sens, c'est-à-dire, l'augmentation de la valeur de la fonction de coût représente une exécution plus longue en temps. Dans 16,6% des cas, une variation nulle de la valeur de la fonction de coût a donné une variation faible (inférieur à 3,0%) du temps d'exécution expérimental. Ainsi, par rapport

au sens de la variation, environ 65% (48,3%+16,6%) des cas sont bons. Ensuite, nous voulons vérifier si les degrés de variation de la valeur de la fonction de coût et du temps d'exécution sont proches. Nous avons établi le rapport entre la variation des valeurs de la fonction de coût et la variation des temps d'exécution. Supposons $fc_1$ et $fc_2$ deux valeurs de la fonction de coût obtenues. Supposons aussi $tex_1$ et $tex_2$ les temps d'exécution mesurés correspondants. Nous avons calculé le rapport suivant pour toute paire de placements :

$$\frac{\frac{|fc_1 - fc_2|}{fc_1}}{\frac{|tex_1 - tex_2|}{tex_1}}$$

Si le rapport ci-dessus vaut 1, une variation des valeurs de la fonction de coût est répercutée à la même proportion dans les temps mesurés. Le Tableau 6.4 présente les rapports obtenus pour les variations qui sont allées dans le même sens (les 48,3% des cas ci-dessus) et aussi pour les variations qui sont allées dans le sens inverse (e.g., une augmentation des valeurs de la fonction de coût et une respective diminution des temps d'exécution). Une conclusion importante est que, en général, une augmentation considérable de la fonction de coût représente une augmentation plus mitigée du temps d'exécution : il y a plus de cas qui ont un rapport supérieur à 1,2.

| rapport entre variations | même sens | sens inverse |
|---|---|---|
| <0,1 | 17 | 11 |
| ≥0,1 et <0,8 | 97 | 82 |
| ≥0,8 et <1,2 | 37 | 31 |
| ≥1,2 et <10,0 | 265 | 154 |
| ≥10,0 | 75 | 82 |

Tableau 6.4 : Les rapports entre les variations des valeurs de la fonction de coût et les variations des temps mesurés.

Enfin, la qualité de la fonction de coût TOR est raisonnablement bonne, mais est-elle coûteuse à calculer? Dans la Figure 6.22, le rapport entre le temps d'exécution des stratégies de placement avec la fonction de coût TOR et le temps d'exécution des stratégies de placement avec la fonction de coût ADD est présenté. Ce rapport a été établi pour chaque stratégie de placement itérative (pour le tabou – `qtabu` dans la Figure 6.22 – et pour le recuit simulé – `qsiman` dans la Figure 6.22). Il peut être significatif dans certains cas.

Pour les stratégies gloutonnes, les temps mesurés étaient toujours au-dessous de 1 seconde pour les deux fonctions de coût, donc cette différence n'est pas appréciable dans le graphe. Les rapports dans la Figure 6.22 sont très variables et cela est causé par l'aspect aléatoire des algorithmes itératifs. L'ordonnée du point qui n'est pas représenté

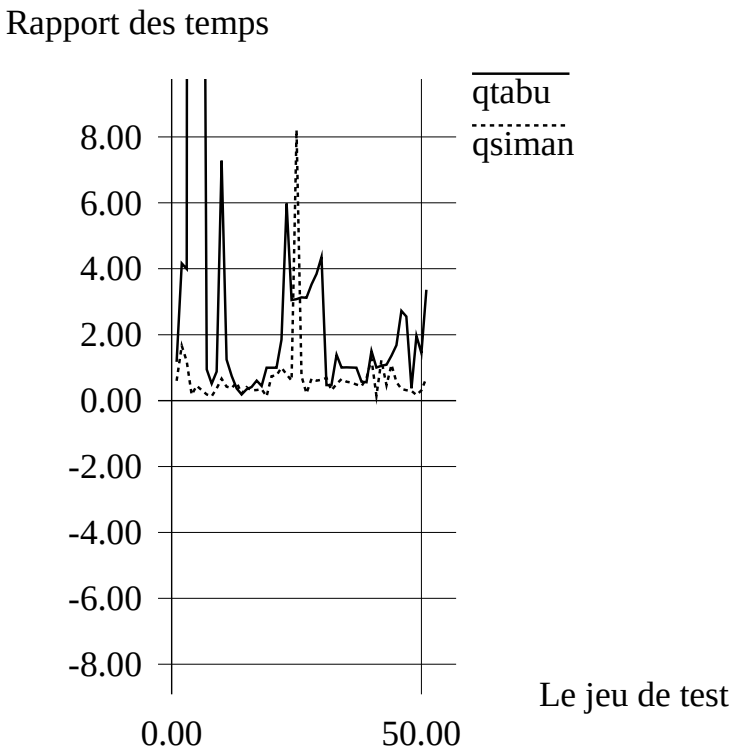



Figure 6.22 : Le rapport entre les temps d'exécution des stratégies de placement.

dans le graphe est 164 et elle correspond au modèle de `DIAMOND1` avec le plus grand rapport communication/calcul. Les tâches de ce modèle ont toutes le même coût de calcul et de communication. Le tri réalisé par la stratégie `quanti` avec la fonction de coût TOR ne sert à rien dans ce cas. Le `qtabu` a beaucoup d'espace pour améliorer le placement original donné par `quanti` et cela prend du temps.

En conclusion, le temps dépensé pour exécuter les stratégies gloutonnes, soit pour la fonction de coût ADD, soit pour la fonction de coût TOR, sont identiques. Pour les stratégies itératives, ils sont, en général, supérieures pour la stratégie itérative `qtabu` à cause du calcul supplémentaire du routage. Pour `qsiman`, cette quantité est moins importante et cela donne des rapports plus petits. Nous insistons sur le fait que les temps mesurés ont une très basse précision (1 seconde) et que les algorithmes itératifs de placement ont un caractère aléatoire. Dans la suite on ne considérera que la fonction de coût TOR.

**Relation entre les paramètres des modèles du jeu de test**. Nous avons calculé les coefficients de corrélation de toutes les paires possibles de paramètres : PV (le

parallélisme virtuel), DM (le degré moyen du graphe groupé), CC (le rapport communication/calcul en temps des groupes), RG (la régularité de communication du *DG-ANDES* groupé) et SUP (l'accélération obtenue). Le Tableau 6.5 présente les coefficients pour les deux jeux de test ensemble (regroupement à la main et automatique). Toutes les mesures ont été utilisées pour la fonction de coût TOR et pour les stratégies de placement `modulo`, `lpt`, `quanti`, `struc_quanti`, `qsiman` et `qtabu`. Les données ont été normalisées afin d'enlever les unités de chaque paramètre (cela correspond, pour chaque paramètre, à soustraire la moyenne de sa valeur et à diviser cette différence par l'écart-type).

| | PV | DM | CC | RG | SUP |
|---|---|---|---|---|---|
| PV | 1,00 | | | | |
| DM | -0,04 | 1,00 | | | |
| CC | 0,08 | -0,03 | 1,00 | | |
| RG | 0,10 | 0,13 | 0,17 | 1,00 | |
| SUP | 0,01 | -0,14 | -0,67 | -0,28 | 1,00 |

Tableau 6.5 : Les coefficients de corrélation linéaires pour les paramètres des graphes groupés (automatiquement et à la main).

Il est intéressant de comparer les coefficients donnés par le Tableau 6.5 et les coefficients calculés pour les accélérations théoriques (au lieu d'utiliser le temps mesuré pour obtenir l'accélération, on utilise la valeur de la fonction de coût). La comparaison est faite dans le Tableau 6.6. La fonction de coût ne prend pas en compte d'autres paramètres (différents du CC) des programmes parallèles qui, dans la pratique, jouent un rôle important pour déterminer l'accélération d'une application parallèle.

| | PV | DM | CC | RG |
|---|---|---|---|---|
| SUP théorique | -0,07 | 0,06 | -0,69 | -0,07 |
| SUP expérimental | 0,01 | -0,14 | -0,67 | -0,28 |

Tableau 6.6 : Les coefficients de corrélation linéaires pour les accélérations théorique et expérimental.

Les conclusions à retenir concernent l'accélération et les autres paramètres. On constate à partir des tableaux que les paramètres les plus fortement liés sont l'accélération et le rapport communication/calcul. Cette relation est attendue : plus il y a de communications, plus il y a de surcharge et moins il y a d'accélération. Dans une mesure plus faible de dépendance, l'accélération dépend de la régularité de communication du

graphe (moins régulière est la communication entre les groupes, moins d'accélération). Les valeurs négatives des coefficients indiquent que les variations respectives de ces deux variables sont de sens inverses.

**Les stratégies de placement statique et les modèles du jeu de test**. Afin de comprendre le rapport entre les accélérations obtenues à partir des mesures, les paramètres des jeux de test et les stratégies de placement statique, une ACP a été réalisée. Les individus sont les 17 modèles du jeu de test avec 2 regroupements différents (à la main et automatique), avec 3 rapports communication/calcul distincts pour chaque modèle et 5 stratégies de placement statique (`lpt`, `quanti`, `struc_quanti`, `qsiman` et `qtabu`). Cela donne 17*2*3*5 (= 510) individus. Les variables sont PV, DM, CC, RG et SUP.

Quelle est la qualité de représentation des nouveaux axes? Les valeurs propres (les inerties) associées à chaque axe sont représentées dans le Tableau 6.7.

| Axes | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Inertie | 1,82 | 1,07 | 1,02 | 0,78 | 0,30 |
| Inertie en % | 36,46 | 21,49 | 20,49 | 15,65 | 5,91 |
| Inertie cumulée | 36,46 | 57,95 | 78,43 | 94,09 | 100 |

Tableau 6.7 : Les valeurs propres obtenues.

Les trois premiers axes représentent bien (78%) le nuage d'individus. Les qualités de représentation des variables sur les nouveaux axes sont données par le Tableau 6.8. Les variables SUP, CC et RG sont mieux représentées sur l'axe 1. La variable DM est mieux représentée sur l'axe 2 et la variable PV est mieux représentée sur l'axe 3.

| Variables | Axe 1 | Axe 2 | Axe 3 |
|---|---|---|---|
| PV | 1% | 2% | 68% |
| DM | 4% | 72% | 3% |
| CC | 70% | 9% | 5% |
| RG | 27% | 7% | 21% |
| SUP | 80% | 0% | 4% |

Tableau 6.8 : Les qualités de représentation des variables sur les axes 1, 2 et 3.

Nous présentons, dans la Figure 6.23, les individus représentés sur les plans définis par les nouveaux axes. Les flèches ($\uparrow$ et $\downarrow$) représentent la variation de la variable en fonction de la croissance ou décroissance des valeurs sur les axes. Par exemple, $\downarrow$ DM pour l'axe 2 indique une décroissance des valeurs de DM en fonction de la croissance

des valeurs de cet axe : les individus dans la région **A** ont un degré moyen plus petit que les individus de la région **C**.

Les graphes de la Figure 6.23 montrent que les variables DM et PV discriminent bien le nuage de points. Les régions dessinées sur les graphes sont composées de :

- sur le plan axe1-axe2 :
  - région **A** : `BELLFORD2`, `DIAMOND1`, `DIAMOND2`, `DIAMOND3`, `DIAMOND4`, `DIVCONQ`, `FFT2`, `GAUSS` (groupé automatiquement et avec le plus grand CC), `ITERATIVE2`, `MS3`, `MS_GAUSS` (groupé automatiquement), `PDE1`, `PDE2`, `PROLOG`, `QCD2` et `STRASSEN2`;
  - région **B** : `MS_GAUSS` (groupé à la main) et `WARSHALL` (groupé automatiquement);
  - région **C** : `GAUSS` (sauf `GAUSS` groupé automatiquement et avec le plus grand CC) et `WARSHALL` (groupé à la main).
- sur le plan axe1-axe3 :
  - région **D** : `FFT2`, `PDE1` et `ITERATIVE2`;
  - région **E** : `DIVCONQ`, `GAUSS`, `PDE2`, `PROLOG`, `STRASSEN2` et `WARSHALL`;
  - région **F** : `BELLFORD2`, `DIAMOND3` et `DIAMOND4`;
  - région **G** : `DIAMOND1`, `DIAMOND2`, `MS_GAUSS`, `MS3` et `QCD2`.

Avant de passer aux conclusions, nous présentons, dans les Tableaux 6.9 et 6.10 :

1. les modèles du jeu de test;
2. les rapports entre les accélérations données par `quanti`, `struc_quanti`, `qsiman` et `qtabu` par rapport à l'accélération donnée par `lpt`;
3. la stratégie qui a donné la meilleure accélération ($acc_{max}$);
4. la stratégie qui a donné la pire accélération ($acc_{min}$);
5. le rapport :

$$MINMAX = \frac{tex_{max} - tex_{min}}{tex_{min}}$$

où $tex_{max}$ est la valeur du meilleur temps d'exécution obtenu et $tex_{min}$ est la valeur du pire temps d'exécution obtenu. Pour $MINMAX$, toutes les stratégies sont prises en compte (`modulo`, `lpt`, `quanti`, `struc_quanti`, `qsiman` et `qtabu`).

### 6.3.4 L'interprétation des résultats

**Le comportement général des stratégies**. La stratégie `modulo` est la pire stratégie dans 52,9% des cas (en considérant tous les tests avec les deux regroupements). `lpt` est la pire stratégie dans 23,5% des cas. Dans 11,7% des cas, `modulo` et `lpt` ne sont pas les pires, mais la différence entre la meilleure stratégie et la pire est plus petite que 10%. Si cette différence est plus grande que 10%, la pire stratégie est itérative (dans un algorithme itératif, un minimum local n'est pas forcément un minimum global). Une autre possibilité est, par exemple, le modèle `MS3` (avec un facteur *c* de communication) dans le Tableau 6.9. La pire stratégie pour ce cas est `quanti`, mais la différence entre `quanti` et `lpt` est nulle. `modulo` n'est pas la pire stratégie dans tous les cas, car :

- par hasard, `modulo` peut donner une bonne solution : plusieurs mesures doivent être faites afin de minimiser l'importance du "hasard" donnant de bonnes solutions;
- lorsque la quantité de communication de l'application est petite et que les groupes ont presque le même coût de calcul, toutes les stratégies basées sur la distribution uniforme de groupes sont bonnes (e.g., `modulo` et `lpt`).

A cause de ce comportement aléatoire de `modulo`, nous avons choisi `lpt` comme la stratégie de base pour la comparaison des accélérations :

- dans 78,4% des cas, `quanti` a été meilleur que `lpt`.
- dans 18,6% des cas, `lpt` a été meilleur que `quanti`, mais à 10% de différence au maximum;
- dans 81,4% des cas, `struc_quanti` a été meilleur que `lpt`;
- dans 15,7% des cas, `lpt` a été meilleur que `struc_quanti`, mais à 10% de différence au maximum;
- dans 76,5% des cas, `qsiman` a été meilleur que `lpt`;
- dans 19,6% des cas, `lpt` a été meilleur que `qsiman`, mais à 10% de différence au maximum;
- dans 75,5% des cas, `qtabu` a été meilleur que `lpt`;
- dans 19,6% des cas, `lpt` a été meilleur que `qtabu`, mais à 10% de différence au maximum.

Dans 75,5% des cas, `quanti`, `struc`, `qsiman` ou `qtabu` est la meilleure stratégie. Si on ne considère pas `modulo`, cette chiffre passe à 81%. Dans les études théoriques sur la fonction de coût, les meilleures accélérations par modèle du jeu de

test sont données par les stratégies qui prennent en compte la communication. Si `lpt` est la meilleure stratégie, elle est à 5% de différence au maximum. Sans considérer `modulo`, `lpt` est la plus mauvaise stratégie dans 67% des cas, contre 83% des cas sur le plan de la fonction de coût. Dans 20% des cas, `lpt` n'est pas la pire stratégie, mais la différence entre la pire et la meilleure est inférieure que 10% (contre 5% donné par les études théoriques). On vérifie que `lpt` se comporte mieux dans la pratique. Lorsque `qsiman` donne une accélération supérieure à `quanti`, l'amélioration n'est jamais plus grande que 5% (sauf pour `GAUSS` groupé automatiquement et à la main et avec un facteur *b* de communication). Lorsque `qtabu` donne une accélération supérieure à `quanti`, l'amélioration n'est jamais plus grande que 10%. Considérant les résultats trouvés en analysant les valeurs de la fonction de coût (évaluation théorique), l'amélioration des algorithmes itératifs par rapport aux gloutons est aussi faible, mais moins faible que dans la pratique. Si dans les expériences, cette amélioration est plus petite que 10%, dans la théorie, 80,5% des cas ont donné une variation plus petite que 10%. Sans considérer `modulo`, on vérifie, sur les plans de la Figure 6.23 et sur les chiffres des Tableaux 6.9 et 6.10, que l'écart entre les temps mesurés pour un modèle du jeu de test (avec un rapport communication/calcul et un regroupement donnés) n'est pas grand. Les points isolés sur les plans sont, en fait, composés de point superposés.

**Le placement et les paramètres du jeu de test**. Nous nous intéressons aux cas qui ont donné des meilleures accélérations. Ces cas sont présentés dans le Tableau 6.11.

Par rapport à la moyenne globale des paramètres (cette moyenne est donnée dans la dernière ligne du Tableau 6.11), les bonnes accélérations sont obtenues :

- avec un PV faible;
- avec un DM faible;
- avec un CC faible;
- avec un RG faible.

Certaines de ces conclusions ont été obtenues avant par le moyen des coefficients de corrélation. Les modèles `DIAMOND1`, `DIAMOND3` et `BELLFORD2` listés dans le Tableau 6.11 présentent un PV, un DM, un CC et un RG faibles. Par contre, `DIVCONQ` est très irrégulier, mais son CC et son DM sont assez faibles pour donner une bonne accélération. Le même est valable pour `PROLOG`. `FFT2` a un fort DM, mais ce modèle est celui qui a le plus grand PV dans le jeu de test. Ce parallélisme virtuel fort doit jouer un rôle important dans l'obtention d'une bonne accélération. Enfin, `MS_GAUSS`, malgré son DM fort, a donné une bonne accélération : son CC est assez faible et il s'agit d'un modèle régulier.

## 6.4 Conclusions

Le but de ce chapitre était de montrer que l'outil *ANDES-Synth* (et derrière cet outil, le modèle *ANDES*) peut être utilisé avec bénéfice dans le cadre d'une étude d'évaluation

des performances. L'exécution des charges synthétiques est la technique employée. Nous avons montré qu'il est possible de jouer avec différents modèles de programmes parallèles (le jeu de test développé), avec différents modèles de machine (en variant la bande passante du réseau) et avec différentes stratégies d'implémentation (de regroupement et de placement statique).

Au-delà de l'exemple d'utilisation, nous sommes arrivé à certaines conclusions sur les stratégies de placement statiques évaluées. Nous avons vérifié plus formellement certaines hypothèses concernant le jeu de test et les stratégies utilisées :

- **la meilleure fonction de coût est celle qui modélise le routage VCR sur le tore 4x4** : l'utilisation de telle fonction n'est pas coûteuse en temps par rapport à la fonction de calcul qui ne tient pas compte du routage;
- TOR est la meilleure fonction par rapport à MAX, ADD et ROUT. Néanmoins, **TOR est une fonction de coût optimiste : une grande variation de ses valeurs ne représentent, en général, une grande variation des temps mesurés**;
- **le rapport communication/calcul – en temps – a une forte influence sur l'exécution parallèle** : nous avons pu le constater par le moyen des coefficients de corrélation linéaire entre ce rapport et l'accélération. Cette importance a été constatée aussi sur le plan théorique;
- **la régularité de communication a une influence moins forte, mais visible quand-même** : cela a été vérifié aussi par le moyen des coefficients de corrélation linéaire entre la régularité de communication du graphe groupé et l'accélération. Sur le plan théorique, ces paramètres ne jouent pas un rôle décisif sur la détermination de l'accélération;
- **les stratégies les plus performantes sont celles qui prennent en compte la communication** : les stratégies `quanti`, `qsiman` et `qtabu` donnent des résultats proches. Cependant, `quanti` s'exécute plus vite, car elle est une stratégie gloutonne. Les stratégies itératives améliorent peu le placement donné par `quanti`. La régularité et la symétrie (soit quantitatives, soit structurales) doivent être la raison ce ce comportement : l'échange de paires de groupes ne change pas la configuration de l'application. Ce comportement a été vérifié sur le plan théorique;
- **`lpt` est une mauvaise stratégie dans la théorie et pas si mauvaise dans la pratique** et **l'écart entre les stratégies est, en général, faible**;
- **en général, pour le jeu de test considéré, les bonnes accélérations sont données par les modèles avec un parallélisme virtuel plus faible que la moyenne**. Néanmoins, les accélérations de `FFT2` sont bonnes (ce modèle présente le plus grand PV du jeu de test).

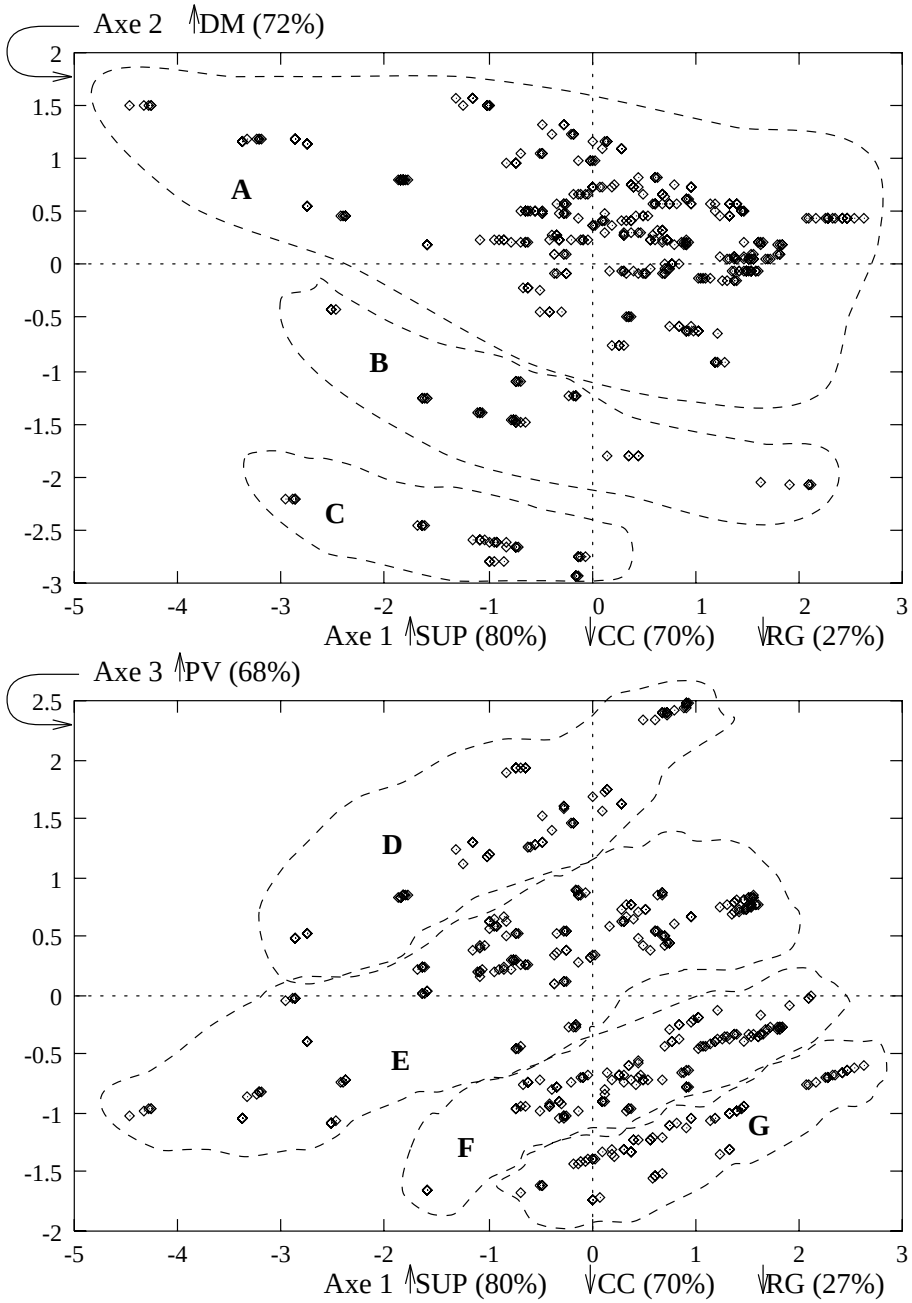



Figure 6.23 : Les individus sur les nouveaux plans.

| modèle | qua-lpt | str-lpt | qsi-lpt | qta-lpt | meil. | pire | MINMAX |
|---|---|---|---|---|---|---|---|
| bell2-a | 1,05 | 1,16 | 1,05 | 1,05 | str | lpt | 0,16 |
| bell2-b | 1,06 | 1,13 | 1,06 | 1,06 | str | mod | 0,14 |
| bell2-c | 1,01 | 1,12 | 1,01 | 1,01 | str | mod | 0,13 |
| diam1-a | 1,11 | 1,11 | 1,11 | 1,11 | str | mod | 0,16 |
| diam1-b | 1,08 | 1,08 | 1,08 | 1,12 | qta | lpt | 0,11 |
| diam1-c | 1,10 | 1,10 | 1,10 | 1,18 | qta | lpt | 0,18 |
| diam2-a | 1,25 | 1,26 | 1,07 | 1,29 | qta | mod | 0,39 |
| diam2-b | 1,14 | 1,14 | 1,00 | 1,04 | str | mod | 0,19 |
| diam2-c | 1,12 | 1,07 | 1,15 | 1,05 | qsi | mod | 0,18 |
| diam3-a | 1,26 | 1,14 | 1,24 | 1,18 | qua | mod | 0,28 |
| diam3-b | 1,17 | 1,09 | 1,17 | 1,19 | qta | lpt | 0,18 |
| diam3-c | 1,13 | 1,13 | 1,02 | 1,13 | str | lpt | 0,13 |
| diam4-a | 1,05 | 1,05 | 1,05 | 1,04 | qsi | mod | 0,15 |
| diam4-b | 1,10 | 1,13 | 1,08 | 1,09 | str | mod | 0,23 |
| diam4-c | 1,05 | 1,09 | 1,05 | 1,05 | str | mod | 0,17 |
| divcq-a | 1,05 | 1,03 | 1,06 | 1,05 | qsi | lpt | 0,06 |
| divcq-b | 1,06 | 1,04 | 1,06 | 1,07 | qta | mod | 0,08 |
| divcq-c | 1,06 | 1,05 | 1,07 | 1,07 | qsi | mod | 0,07 |
| fftr2-a | 1,09 | 1,09 | 1,09 | 1,09 | qta | lpt | 0,08 |
| fftr2-b | 1,11 | 1,10 | 1,11 | 1,11 | qsi | mod | 0,11 |
| fftr2-c | 1,11 | 1,12 | 1,11 | 1,11 | str | mod | 0,12 |
| gauss-a | 1,08 | 1,06 | 0,99 | 1,08 | qua | qsi | 0,08 |
| gauss-b | 1,02 | 1,02 | 1,10 | 0,97 | qsi | qta | 0,13 |
| gauss-c | 1,02 | 1,01 | 1,01 | 1,04 | qta | lpt | 0,04 |
| iter2-a | 1,00 | 1,00 | 1,00 | 0,97 | mod | qta | 0,05 |
| iter2-b | 0,96 | 0,96 | 0,96 | 1,05 | qta | qsi | 0,09 |
| iter2-c | 1,13 | 1,13 | 1,06 | 1,07 | qua | mod | 0,14 |
| ms3-a | 0,80 | 0,80 | 0,80 | 0,82 | lpt | qsi | 0,24 |
| ms3-b | 0,74 | 0,74 | 0,74 | 0,74 | lpt | mod | 0,35 |
| ms3-c | 0,72 | 0,72 | 0,72 | 0,80 | lpt | qua | 0,38 |
| msgas-a | 1,11 | 1,22 | 1,11 | 1,11 | str | lpt | 0,21 |
| msgas-b | 1,06 | 1,09 | 1,06 | 1,06 | str | lpt | 0,08 |
| msgas-c | 0,98 | 1,03 | 0,98 | 0,98 | str | mod | 0,09 |
| pde01-a | 1,24 | 1,22 | 1,22 | 1,24 | qua | mod | 0,35 |
| pde01-b | 1,18 | 1,15 | 1,18 | 1,18 | qta | mod | 0,22 |
| pde01-c | 1,13 | 1,13 | 1,14 | 1,13 | qsi | mod | 0,20 |
| pde02-a | 1,06 | 1,05 | 1,06 | 1,06 | qsi | mod | 0,18 |
| pde02-b | 1,07 | 1,04 | 1,07 | 1,07 | qua | mod | 0,19 |
| pde02-c | 1,02 | 1,03 | 1,02 | 1,02 | str | mod | 0,19 |
| prolg-a | 1,01 | 1,02 | 1,01 | 1,01 | mod | lpt | 0,02 |
| prolg-b | 1,08 | 1,10 | 1,01 | 1,08 | str | mod | 0,14 |
| prolg-c | 1,10 | 1,08 | 1,10 | 1,10 | mod | lpt | 0,24 |
| qcd02-a | 1,00 | 1,00 | 1,01 | 1,00 | mod | qta | 0,01 |
| qcd02-b | 0,99 | 0,99 | 0,97 | 0,99 | mod | qsi | 0,05 |
| qcd02-c | 0,98 | 0,98 | 0,98 | 0,98 | lpt | qsi | 0,02 |
| stra2-a | 1,26 | 1,25 | 1,25 | 1,18 | qua | mod | 0,27 |
| stra2-b | 1,14 | 1,16 | 1,09 | 1,14 | str | mod | 0,24 |
| stra2-c | 1,10 | 1,06 | 1,11 | 1,10 | qsi | mod | 0,14 |
| warsh-a | 0,93 | 0,99 | 0,94 | 0,93 | lpt | mod | 0,08 |
| warsh-b | 0,98 | 0,95 | 0,98 | 0,98 | lpt | mod | 0,09 |
| warsh-c | 0,99 | 0,97 | 0,99 | 0,99 | lpt | mod | 0,03 |

Tableau 6.9 : Les rapports des accélérations (regroupement automatique).

| modèle | qua-lpt | str-lpt | qsi-lpt | qta-lpt | meil. | pire | MINMAX |
|---|---|---|---|---|---|---|---|
| bell2-a | 1,07 | 1,07 | 1,07 | 1,07 | qsi | mod | 0,15 |
| bell2-b | 1,04 | 1,01 | 0,99 | 1,02 | qua | mod | 0,08 |
| bell2-c | 1,07 | 1,05 | 1,06 | 1,02 | qua | lpt | 0,07 |
| diam1-a | 1,10 | 1,10 | 1,11 | 1,10 | qsi | mod | 0,23 |
| diam1-b | 1,07 | 1,08 | 1,08 | 1,10 | qta | mod | 0,10 |
| diam1-c | 1,08 | 1,08 | 1,08 | 1,11 | qta | mod | 0,11 |
| diam2-a | 1,21 | 1,22 | 1,22 | 1,22 | str | mod | 0,42 |
| diam2-b | 1,13 | 1,13 | 1,13 | 1,14 | qta | mod | 0,15 |
| diam2-c | 1,16 | 1,17 | 1,17 | 1,18 | qta | mod | 0,23 |
| diam3-a | 1,27 | 1,21 | 1,27 | 1,30 | qta | mod | 0,30 |
| diam3-b | 1,09 | 1,07 | 1,09 | 1,07 | qsi | lpt | 0,08 |
| diam3-c | 1,09 | 1,08 | 1,09 | 1,07 | qsi | mod | 0,13 |
| diam4-a | 1,05 | 1,07 | 1,05 | 1,04 | mod | lpt | 0,07 |
| diam4-b | 1,24 | 1,24 | 1,24 | 1,23 | str | lpt | 0,23 |
| diam4-c | 1,18 | 1,15 | 1,17 | 1,13 | qua | lpt | 0,18 |
| divcq-a | 1,06 | 1,09 | 1,06 | 1,06 | str | lpt | 0,09 |
| divcq-b | 1,08 | 1,06 | 1,04 | 1,08 | qta | mod | 0,10 |
| divcq-c | 1,07 | 1,06 | 1,07 | 1,07 | qsi | mod | 0,08 |
| fftr2-a | 1,02 | 1,02 | 1,02 | 1,02 | qsi | mod | 0,03 |
| fftr2-b | 1,07 | 1,07 | 1,07 | 1,07 | str | mod | 0,11 |
| fftr2-c | 1,06 | 1,06 | 1,06 | 1,05 | qua | mod | 0,08 |
| gauss-a | 1,04 | 1,03 | 1,02 | 0,84 | mod | qta | 0,32 |
| gauss-b | 0,98 | 0,98 | 1,08 | 0,96 | qsi | qta | 0,12 |
| gauss-c | 0,99 | 1,00 | 1,00 | 0,99 | lpt | mod | 0,05 |
| iter2-a | 1,00 | 1,00 | 1,00 | 0,97 | lpt | qta | 0,03 |
| iter2-b | 0,96 | 0,96 | 0,96 | 0,96 | mod | qta | 0,05 |
| iter2-c | 1,13 | 1,13 | 1,02 | 1,07 | qua | lpt | 0,13 |
| ms3-a | 1,00 | 1,00 | 1,00 | 1,07 | qta | str | 0,07 |
| ms3-b | 1,00 | 1,00 | 0,97 | 1,05 | mod | qsi | 0,11 |
| ms3-c | 1,00 | 1,00 | 0,94 | 1,01 | qta | qsi | 0,07 |
| msgas-a | 0,96 | 1,03 | 0,96 | 0,96 | str | mod | 0,34 |
| msgas-b | 0,94 | 0,99 | 0,82 | 0,94 | lpt | qsi | 0,21 |
| msgas-c | 0,98 | 0,83 | 0,98 | 0,92 | lpt | mod | 0,26 |
| pde01-a | 1,15 | 1,15 | 1,15 | 1,15 | qta | mod | 0,28 |
| pde01-b | 1,17 | 1,17 | 1,16 | 1,17 | str | mod | 0,27 |
| pde01-c | 1,10 | 1,10 | 1,09 | 1,10 | str | mod | 0,23 |
| pde02-a | 1,10 | 1,10 | 1,12 | 1,13 | qta | mod | 0,17 |
| pde02-b | 1,11 | 1,11 | 1,11 | 1,12 | qta | mod | 0,20 |
| pde02-c | 1,10 | 1,10 | 1,11 | 1,09 | qsi | mod | 0,17 |
| prolg-a | 0,96 | 0,96 | 0,96 | 0,96 | mod | qua | 0,04 |
| prolg-b | 1,07 | 1,07 | 0,97 | 1,07 | qua | qsi | 0,10 |
| prolg-c | 1,12 | 1,12 | 1,12 | 1,12 | str | lpt | 0,11 |
| qcd02-a | 0,90 | 0,90 | 0,90 | 0,89 | mod | qta | 0,14 |
| qcd02-b | 0,95 | 0,95 | 0,95 | 0,95 | mod | str | 0,15 |
| qcd02-c | 1,08 | 1,08 | 1,06 | 1,05 | str | lpt | 0,08 |
| stra2-a | 1,66 | 1,44 | 1,62 | 1,65 | qua | lpt | 0,65 |
| stra2-b | 1,24 | 1,15 | 1,21 | 1,24 | qta | lpt | 0,24 |
| stra2-c | 1,09 | 1,14 | 1,07 | 1,09 | str | lpt | 0,14 |
| warsh-a | 0,98 | 0,98 | 0,98 | 0,90 | lpt | qta | 0,10 |
| warsh-b | 0,99 | 0,99 | 1,03 | 0,92 | qsi | qta | 0,12 |
| warsh-c | 0,98 | 0,98 | 0,98 | 0,91 | lpt | qta | 0,09 |

Tableau 6.10 : Les rapports des accélérations (regroupement manuel).

| modèle | PV | DM | CC | RG |
|---|---|---|---|---|
| diamond1 | 15.78 | 1.00 | 0.06 | 0.25 |
| diamond1 | 15.78 | 1.00 | 0.02 | 0.25 |
| diamond3 | 33.40 | 2.66 | 0.12 | 0.45 |
| diamond3 | 33.40 | 2.26 | 0.04 | 0.45 |
| divconq | 80.74 | 1.99 | 0.05 | 0.83 |
| divconq | 80.74 | 1.99 | 0.04 | 0.83 |
| divconq | 80.74 | 1.99 | 0.03 | 0.83 |
| fft2 | 426.83 | 6.61 | 0.23 | 0.51 |
| fft2 | 426.83 | 6.61 | 0.15 | 0.51 |
| prolog | 77.08 | 1.98 | 0.06 | 0.82 |
| bellford2-gr | 28.08 | 3.62 | 0.06 | 0.45 |
| diamond1-gr | 15.78 | 1.00 | 0.39 | 0.25 |
| diamond1-gr | 15.78 | 1.00 | 0.06 | 0.25 |
| diamond1-gr | 15.78 | 1.00 | 0.02 | 0.25 |
| diamond3-gr | 33.40 | 1.80 | 0.12 | 0.45 |
| diamond3-gr | 33.40 | 1.80 | 0.04 | 0.45 |
| divconq-gr | 80.74 | 1.00 | 0.08 | 0.90 |
| divconq-gr | 80.74 | 1.00 | 0.06 | 0.90 |
| divconq-gr | 80.74 | 1.00 | 0.04 | 0.90 |
| fft2-gr | 426.83 | 9.48 | 0.22 | 0.43 |
| fft2-gr | 426.83 | 9.48 | 0.14 | 0.43 |
| ms_gauss-gr | 25.71 | 15.59 | 0.09 | 0.19 |
| **MOYENNE DE LA POPULATION** | **98,72** | **4,69** | **0,63** | **0,58** |

Tableau 6.11 : Les paramètres des modèles qui ont donné les meilleures accélérations.

# Chapitre 7

# Conclusions et Perspectives

## 7.1 Conclusions

Cette thèse représente l'aboutissement de trois années de recherche dans le domaine de la modélisation quantitative d'algorithmes parallèles. Elle est liée aussi à d'autres domaines comme les programmes synthétiques, la modélisation quantitative de machines parallèles, les problèmes du placement et du regroupement.

La littérature est vaste sur les modèles d'algorithmes et de programmes parallèles. La complexité de ces modèles varie selon les objectifs des utilisations. Cependant, les types de modèles quantitatifs pour l'évaluation des performances ne sont pas nombreux. En général, ce sont des graphes orientés ou non-orientés avec des attributs numériques représentant les coûts de calcul, de communication et de mémoire. Pour certains contextes de recherche, ces modèles sont suffisants. Néanmoins, certains aspects importants des algorithmes et des programmes parallèles ne sont pas modélisés par ces types de graphes. L'exemple typique est l'influence des données sur le flot de contrôle. Avec *ANDES*, nous permettons une modélisation plus flexible de l'influence de données. Nous croyons que, dans un contexte d'évaluation des performances, cette modélisation est capitale.

Les modèles *ANDES* sont exprimés en utilisant la bibliothèque *ANDES-C*. L'expression textuelle qui caractérise les modèles décrits en *ANDES-C* est moins parlante que l'expression graphique ou visuelle, mais plus puissante pour la représentation de modèles avec un nombre massif de nœuds de calcul. L'utilisation d'un langage de programmation pour la construction de la bibliothèque *ANDES-C* nous a aidé principalement pour la représentation de modèles avec une structure régulière et pour la génération de charges synthétiques.

L'outil *ANDES-Synth* est employé pour l'évaluation des performances de systèmes informatiques parallèles. La technique d'évaluation est l'analyse de mesures générées par une charge synthétique associée à un modèle *ANDES* d'algorithme parallèle. Ce modèle *ANDES* (exprimé par le moyen de *ANDES-C*) auquel on ajoute un modèle de machine et un ensemble de stratégies d'implémentation permet de générer une charge

synthétique. Cette charge s'exécute sur une machine cible (actuellement, le Méganode). L'atout de cette technique est qu'elle produit un environnement proche de la réalité tout en gardant la flexibilité des modèles. Le désavantage est qu'elle est chère, car une machine cible et un logiciel adéquat doivent être disponibles. Dans une première étape, l'outil a été utilisé pour l'évaluation des stratégies de placement statique.

Nous insistons sur le fait que, dans la littérature, nous n'avons pas trouvé de démarche similaire.

Les conclusions tirées concernent le modèle *ANDES* et son utilisation dans un environnement d'évaluation des performances basé sur les charges synthétiques (*ANDES-Synth*). Le modèle est très adéquat pour la modélisation du flot de contrôle des algorithmes parallèles. Nous n'avons pas eu de grandes difficultés pour représenter, par exemple, des programmes parallèles basés sur le modèle par passage de messages. Par contre, les algorithmes ayant une utilisation particulière de la mémoire (soit par virtualisation et partage, soit par allocation et libération) même si *ANDES* prévoit un certain support pour cela ne sont pas facile à décrire avec la plate-forme construite. Nous ne pouvons pas, pour le moment, nous détacher de l'influence de la mémoire distribuée présente dans notre environnement : l'utilisation d'une mémoire virtualisée distribuée balbutie encore. La modélisation des aspects dynamiques des algorithmes présente une problématique similaire, mais quand-même plus familière à cause de l'intense recherche sur les systèmes d'équilibrage dynamique de charge (cf. Athapascan [Chr94]). L'utilisation d'un langage de programmation hôte pour la représentation de modèles *ANDES* s'est montrée satisfaisante pour un prototype. Dans cette première étape, cette représentation nous a permis de discerner les éléments de base d'un modèle *ANDES* (nœuds de calcul, logiques d'entrée et de sortie, précédence). La conception de la bibliothèque `andes.h` aurait pu être plus rapide et plus élégante avec un langage orienté à objets.

L'environnement *ANDES-Synth* a bien accompli ses objectifs en montrant qu'une machine parallèle peut être utilisée elle aussi comme un outil d'évaluation flexible tout en gardant une certaine fidélité par rapport à la réalité. L'implémentation du gestionnaire d'exécution de la charge synthétique est simple, efficace et fiable. Des dizaines de milliers d'exécutions ont été faites sans occurrence d'erreurs. Les différents temps d'exécution mesurés sont cohérents par rapport à la variation des caractéristiques des charges synthétiques employées. Néanmoins, une difficulté sensible a été l'utilisation d'un modèle de machine décrit de façon centralisée.

Finalement, nous avons vérifié qu'il y a beaucoup de méthodes d'analyse de données qui sont utiles et qui doivent être utilisées. En général, on utilise des régressions linéaires ou de graphiques bidimensionnels. Si les mesures dépendent d'une multitude de paramètres, ces outils ne sont pas suffisants. Dans ce travail de thèse, nous essayons d'aller un peu plus loin par rapport aux approches classiques d'analyse de mesures obtenues d'un ordinateur (parallèle ou non). Le livre de Jain [Jai91] évoque l'importance de ces méthodes. Malheureusement, peu de logiciels *ad hoc* sont couramment disponibles.

## 7.2 Perspectives

Les perspectives de ce travail sont vastes et concernent principalement le traitement des inconvénients repérés pendant le travail de cette thèse. Nous continuerons à travailler dans ce domaine et à développer la puissance du modèle et de l'outil.

Concernant la disponibilité de l'outil, *ANDES-Synth* doit migrer vers la machine cible SP/1 de IBM. Plus que la disponibilité de cet équipement, cette migration représentera un pas vers la portabilité d'outil, car le logiciel utilisé pour son développement doit être standard (PVM – *Parallel Virtual Machine* [SGDM94] – ou MPI – *Message Passing Interface* [CC94]).

Concernant l'évolution de l'outil :

- une nouvelle bibliothèque *ANDES-C* doit être développée en utilisant un langage orienté à objets (C++) afin de pouvoir utiliser des types abstraits associés aux objets de type graphe;
- le modèle de machine développé par Cécile Tron [TP94] au sein du projet APACHE doit être intégré, spécialement les modèles de communication qui prennent en compte un réseau chargé. D'autres aspects doivent être modélisés comme le type et la taille de la mémoire et ses méthodes d'accès;
- l'outil doit être modifié afin de permettre l'évaluation d'autres types de stratégies d'implémentation (ordonnancement et régulation de la charge).

Concernant l'utilisation de l'outil :

- on aura un jeu de test de modèles plus complet afin de permettre la généralisation de conclusions obtenues avec l'outil;
- on fera l'évaluation de nouvelles stratégies d'implémentation permises par l'outil avec un traçage plus fin en utilisant un traceur. De cette façon, d'autres informations que le temps d'exécution pourront être obtenues;
- on testera la robustesse des stratégies de placement statique vis-à-vis des coûts aléatoires des modèles. L'évaluation réalisée dans le chapitre 6 ne concerne que de modèles d'algorithmes avec des coûts constants. On testera aussi si ces stratégies sont extensibles, c'est-à-dire, restent valables pour un nombre plus grand ou plus petit de processeurs;
- il faut réaliser une **validation** complète de notre approche. Il est nécessaire de comparer les exécutions des charges synthétiques issues d'un modèle du programme $x$ avec des exécutions du programme réel $x$. Nous avons démarré cette validation en comparant l'exécution de la charge synthétique associée au modèle `BELLFORD2` (voir le chapitre 6) avec l'exécution d'un programme parallèle (écrit en Inmos C) implémentant l'algorithme de Bellman-Ford. Les

deux exécutions ont réagi de façon identique face aux stratégies de placement. A part les algorithmes de placement, aucun paramètre n'a été changé dans cette expérience. Il faut donc raffiner cette validation;

- on réfléchira sur l'intégration de l'outil *ANDES-Synth* dans un environnement d'aide à la programmation parallèle. L'intégration de notre environnement est même proposée par d'autres travaux décrits dans la littérature, spécialement l'outil PARSE [GCJ94].

Plusieurs articles ont été publiés dans le contexte de cette thèse. Les outils et les résultats de ce travail ont aussi été utilisés par d'autres chercheurs, principalement au sein du projet APACHE (notre travail a été utilisé dans l'évaluation de stratégies statiques et dynamiques de placement, dans l'estimation de la perturbation d'un moniteur pour la mise au point de programmes parallèles). Les articles concernant cette thèse sont disponibles par ftp anonyme dans `imag.fr` (nœud Internet 129.88.30.1) dans le répertoire `pub/APACHE/ALPES`. Les sources de l'outil *ANDES-Synth* peuvent être demandés à l'auteur de cette thèse via courrier électronique (adresse : `kitajima@imag.fr`).

Transputer et Occam2 sont des marques Inmos. Méganode est une marque Telmat. Sparc Sun-4 est une marque Sun Microsystems Inc.

# Appendix A

# Le Méganode

La machine parallèle utilisée actuellement dans l'environnement *ALPES* est le Méganode. Cet appendice a pour but présenter cette machine et l'environnement employé pour la programmation d'*ANDES-Synth*.

## A.1 Le matériel

Le Méganode [Nic89] [MW90] est la machine parallèle conçue dans le projet européen Esprit P1085 "Supernode". Ce multiprocesseur MIMD (à flot multiple d'instructions et à flot multiple de données - en anglais *Multiple Instruction stream, Multiple Data stream*) à mémoire distribuée est commercialisé par Telmat, en France, et par Parsys, au Royaume-Uni. Son microprocesseur de base est le Transputer T800 [Lim92] produit par Inmos et développé aussi dans le projet Esprit 1085. Le T800 (Figure A.1) est doté d'une unité de calcul à virgule flottante de haute vitesse et d'une mémoire interne avec 4 Kilooctets. Le but du projet P1085 était la construction d'un ordinateur extensible capable d'atteindre des performances dans l'ordre du Gigaflops (1 milliard d'opérations virgule flottante par seconde). L'interconnexion entre les processeurs est reconfigurable, c'est-à-dire, toute topologie de degré 4 (nombre de liens bidirectionnels sortant de chaque processeur) est faisable. Cette capacité est dû à l'existence de commutateurs qui sont modifiables avant ou pendant l'exécution d'un programme sur la machine.

L'exemplaire utilisé dans le cadre de cette thèse est le Méganode du Laboratoire de Modélisation et Calcul de l'Institut d'Informatique et de Mathématiques Appliquées de Grenoble (LMC/IMAG), France. Cet exemplaire possède 128 processeurs rangés en 4 modules (en anglais *tandems*) de 32 Transputers chacun (Figure A.2). Chaque module est composé de deux ensembles de 16 Transputers (ensemble connu comme T.Node/16), un dit "maître" et l'autre "esclave". Chaque module a deux commutateurs (en anglais *crossbar*), un pour chaque T.Node/16. Pour réaliser l'interconnexion entre les 4 modules, un commutateur de deuxième niveau est disponible. C'est grâce à cet ensemble de commutateurs (inter-module et intra-module) que le Méganode peut réaliser toute topologie de degré maximal 4 ou moins. Le multiprocesseur a comme

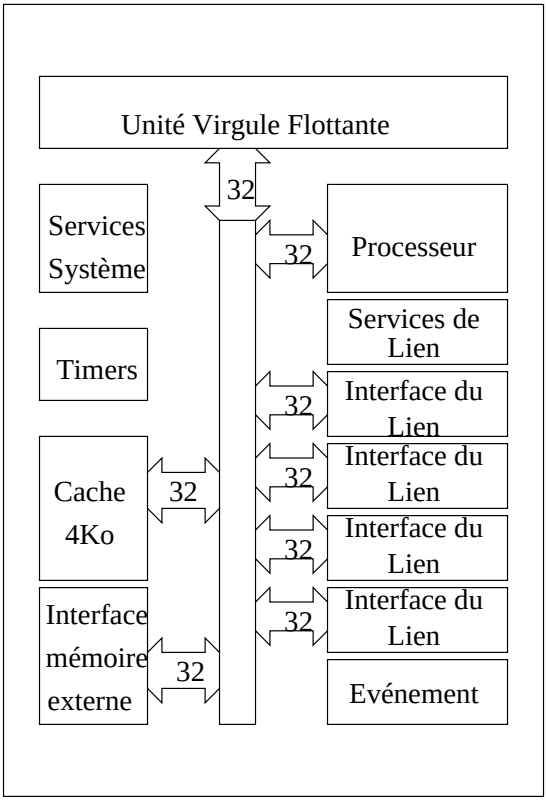



Figure A.1 : Le Transputer T800 [Lim92].

hôte une station de travail Sun-4. Dans l'hôte, il y a une carte à 4 Transputers, chacun travaillant à 20 Mégahertz et ayant 4 Mégaoctets de mémoire. Un de ces processeurs est connecté au commutateur de deuxième niveau du Méganode. Les autres sont utilisés pour le développement de programmes (compilation, édition de liens, etc…).

Chaque processeur du réseau travaille à 20 Mégahertz et les liens ont une bande passante à 10 Mégabits par seconde. Pourtant, les performances de communication entre deux processeurs ne sont pas les mêmes si ces processeurs sont dans un même module ou dans un module distinct. La bande passante du lien entre deux Transputers situés dans un même module est de 1,125 Microsecondes/octet et de 2,2 Microsecondes/octet s'ils sont situés dans des modules différents [TP90]. Il faut finalement remarquer que la mémoire disponible dans chaque processeur du réseau est de 1 Mégaoctet.

## A.2 Le logiciel

Le Méganode du LMC n'a pas un système d'exploitation complet. Actuellement, seulement le chargeur-serveur (le programme *iserver*) et un routeur (VCR - *Virtual*

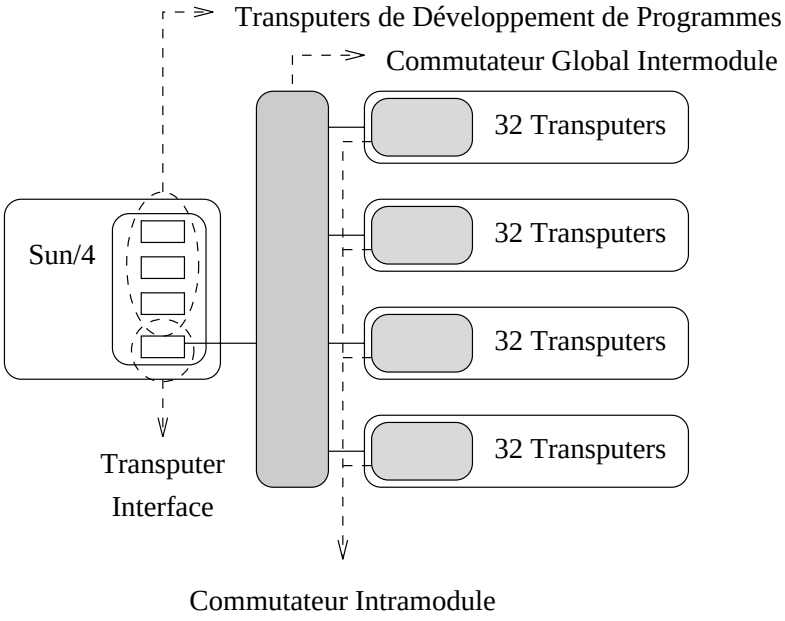



Figure A.2 : Le Méganode du LMC-IMAG.

*Channel Router*) [DHN91] sont disponibles. Le chargeur fait partie du logiciel de base fourni par Inmos et Telmat. Le routeur a été développé par l'Université de Southampton, dans le cadre du projet PUMA. Au niveau de langage, les compilateurs disponibles sont Occam2, Inmos C, 3L C et Logical C. Le modèle de programmation de ces langages est l'échange de messages avec la possibilité de faire de la multiprogrammation sur chaque processeur. VCR est compatible avec Occam2 et Inmos C. A part ces compilateurs de base, deux langages ont été développés au LGI/LMC-IMAG : OUF, basé sur l'échange de messages, et une version primitive d'Athapascan, un langage basé sur les appels de procédures à distance.

# Références


[A+93] André (Françoise) et al. – Programmation des machines à mémoire distribuée par distribution de données: langages et compilateurs. *Technique et Science Informatiques*, vol. 12, nº 5, 1993, pp. 563–596.

[ABZ92] Ambler (Allen L.), Burnett (Margaret M.) et Zimmerman (Betsy A.). – Operational versus definitional: a perspective on programming paradigms. *IEEE Computer*, vol. 25, nº 9, septembre 1992, pp. 28–43.

[AM90] Al-Mouhamed (Mayez A.). – Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. *IEEE Transactions on Software Engineering*, vol. SE-16, nº 12, décembre 1990, pp. 1390–1401.

[Arr92] Arrouye (Yves). – *Ouf/Trita Reference Manual*. – Laboratoire de Modélisation et Calcul-IMAG, INPG, Grenoble, France, octobre 1992. Adresse électronique : arrouye@imag.fr.

[ASU86] Aho (Alfred V.), Sethi (Ravi) et Ullman (Jeffrey D.). – *Compilers: principles, techniques, and tools*. – Reading, Addison-Wesley, 1986, *Addison-Wesley Series in Computer Science*.

[B+92] Balbo (Gianfranco) et al. – An example of modeling and evaluation of a concurrent program using colored stochastic Petri nets: Lamport's fast mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, nº 2, mars 1992, pp. 221–240.

[BA92] Bultan (Tevfik) et Aykanat (Cevdet). – A new mapping heuristic based on mean field annealing. *Journal of Parallel and Distributed Computing*, vol. 16, nº 4, décembre 1992, pp. 292–305.

[Bab84] Babb (R. G.). – Parallel processing with large-grain data-flow techniques. *IEEE Computer*, vol. 17, nº 7, juillet 1984, pp. 55–61.

[Ber94] Bernard (Stéphane). – Notions d'analyse en composantes principales et application au choix de variables. *Tribunix - Dossier Benchmarks*, vol. 10, nº 55, mai 1994, pp. 88–100.

[BJP89] Beck (Micah), Johnson (Richard) et Pingali (Keshav). – *From control flow to dataflow*. – Rapport Technique n° TR 89-1050, Ithaca, Department of Computer Science, Cornell University, octobre 1989.

[BL90] Baccelli (François) et Liu (Zhen). – On the execution of parallel programs on multiprocessor systems – a queueing theory approach. *Journal of the Association Computer Machinery*, vol. 37, n° 2, avril 1990, pp. 373–414.

[BM81] Bondy (J. A.) et Murty (U. S. R.). – *Graph theory with applications*. – New York, North-Holland, 1981.

[Bok81] Bokhari (Shahid H.). – On the mapping problem. *IEEE Transactions on Computers*, vol. C-30, n° 3, mars 1981, pp. 207–214.

[Bou94] Bouvry (Pascal). – *Placement de Tâches sur Machine Parallèle à Mémoire Distribuée*. – Thèse de Doctorat, Institut National Polytechnique de Grenoble, 1994. Directeur de thése : Denis Trystram.

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

[BTV93] Bouvry (Pascal), Trystram (Denis) et Vincent (François). – Placement de tâches sur architectures parallèles à mémoire distribuée - partie 1 : état de l'art. – 1993. Internal Report – LMC/IMAG, Grenoble, France.

[CC94] Calvin (Christophe) et Colombet (Laurent). – *Introduction au Modèle de Programmation par Processus Communicants : Deux Exemples PVM et MPI*. – Rapport Apache 12, Grenoble, IMAG, juillet 1994.

[CG72] Coffman Jr. (E. G.) et Graham (R. L.). – Optimal scheduling for two-processor systems. *Acta Informatica*, vol. 1, n° 3, 1972, pp. 200–213.

[Chi92] Chiola (Giovanni). – GreatSPN 1.5 software architecture. *In : Computer Performance Evaluation: Modelling Techniques and Tools*, éd. par Balbo (Gianfranco) et Serazzi (Giuseppe). pp. 121–136. – Amsterdam, 1992.

[Chr94] Christaller (Michel). – *Athapascan-0a sur PVM3 - Définition et Mode d'Emploi*. – Rapport Apache 11, Grenoble, IMAG, juillet 1994.

[CP92] Candlin (Rosemary) et Phillips (Joe). – An environment for investigating the effectiveness of process migration strategies on Transputer-based machines. *In : Transputer Systems - Ongoing Research/WoTUG-15*, éd. par Allen (Alastair). pp. 13–23. – Amsterdam, 1992.

[CS92] Candlin (Rosemary) et Skilling (Neil). – A modelling system for the investigation of parallel program performance. *In : Computer Performance Evaluation: Modelling Techniques and Tools*, éd. par Balbo (Gianfranco) et Serazzi (Giuseppe). pp. 397–409. – Amsterdam, 1992.

[CT89] Carpenter (Geoffrey F.) et Tyrrell (Andrew M.). – The use of GMB in the design of robust software for distributed systems. *Software Engineering Journal*, septembre 1989, pp. 268–282.

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

[DHN91] Debbage (Mark), Hill (Mark) et Nicole (Denis). – *Virtual Channel Router – Version 2.0 User Guide*. – Department of Electronics & Computer Science, University of Southampton, juin 1991. ftp: `ftp.ecs.soton.ac.uk` (152.78.64.201).

[dJ93] de Jong (G. G.). – *Generalized Data Flow Graphs: Theory and Applications*. – Thèse de Doctorat, Eindhoven University of Technology, 1993. ISBN 90-9006384-6.

[DSN88] Demeure (Isabelle M.), Smith (Sharon L.) et Nutt (Gary J.). – *Modeling parallel, distributed computations using ParaDiGM – a case study: the adaptive global optimization algorithm*. – Rapport Technique nº CU-CS-419-88, Boulder, Department of Computer Science, University of Colorado, décembre 1988.

[E+86] Estrin (Gerald) et al. – SARA (System ARchitects Apprentice): modeling, analysis, and simulation support for design of concurrent systems. *IEEE Transactions on Software Engineering*, vol. SE-12, nº 2, février 1986, pp. 293–311.

[ERL88] El-Rewini (H.) et Lewis (T.). – *Software Development in Parallax: the ELGDF Language*. – Rapport Technique nº 88-60-17, Department of Computer Science, Oregon State University, juillet 1988.

[FB89] Fernández-Baca (David). – Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, vol. SE-15, nº 11, novembre 1989, pp. 1427–1436.

[FJ94] Ferscha (A.) et Johnson (J.). – Performance oriented development of SPMD programs based on task structure specifications. *In : Proceedings of CONPAR 94/VAPP VI - Vienna, Austria*, éd. par Buchberger (Bruno) et Volkert (Jens). pp. 53–65. – Berlin, septembre 1994. Lecture Notes in Computer Science 854.

[FP89] Fdida (Serge) et Pujolle (Guy). – *Modèles de systèmes et de réseaux. Tome 1 : performance*. – Paris, Editions Eyrolles, 1989.

[GCJ94] Gorton (Ian), Chan (Toong Shoon) et Jelly (Innes). – Engineering high quality parallel software using PARSE. *In : Proceedings of CONPAR 94/VAPP VI - Vienna, Austria*, éd. par Buchberger (Bruno) et Volkert (Jens). pp. 381–392. – Berlin, septembre 1994. Lecture Notes in Computer Science 854.

[Gel89] Gelenbe (Erol). – *Multiprocessor performance*. – Chichester, John Wiley and Sons, 1989, *Wiley Series in Parallel Computing*.

[GJ79] Garey (Michael R.) et Johnson (David S.). – *Computers and intractability: a guide to the theory of NP-completeness*. – New York, W. H. Freeman and Company, 1979, *Books in the Mathematical Sciences*.

[GL90] Gillies (Donald W.) et Liu (Jane W.-S.). – Scheduling tasks with AND/OR precedence constraints. *In : Proceedings of the IEEE Symposium on Parallel and Distributed Processing*. pp. 394–406. – Los Alamitos, décembre 1990.

[Glo90] Glover (Fred). – Tabu search: a tutorial. *Interfaces*, vol. 20, n° 4, juillet 1990, pp. 74–94.

[GY92] Gerasoulis (Apostolos) et Yang (Tao). – A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, vol. 16, n° 4, décembre 1992, pp. 276–291.

[H+89] Hwang (J.-J.) et al. – Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, vol. 18, n° 2, avril 1989, pp. 244–257.

[Har88] Harel (David). – On visual formalisms. *Communications of the ACM*, vol. 31, n° 5, mai 1988, pp. 514–530.

[HE91] Heath (Michael T.) et Etheridge (Jennifer A.). – Visualizing the performance of parallel programs. *IEEE Software*, vol. 8, n° 5, septembre 1991, pp. 29–39.

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

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

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

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

[JBG90] Jiang (Hong), Bhuyan (Laxmi N.) et Ghosal (Dipak). – Approximate analysis of multiprocessing task graphs. *In : 1990 International Conference on Parallel Processing*, éd. par Yew (Pu-Chung). pp. III/228–III/235. – University Park, août 1990.

[JM92] Johnen (Colette) et Mourlin (Fabrice). – Analysis of the communication structure of Occam2 programs using Petri nets. *In : Transputers '92: Advanced Research and Industrial Applications*, éd. par Becker (Monique), Litzler (Luc) et Tréhel (Michel). pp. 47–64. – Amsterdam, mai 1992.

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

[KL90] Kim (Inkyu) et Lewis (Ted G.). – *Parallax: an implementation of ELGDF (Extended Large Grain Data Flow)*. – Rapport Technique ń 90-60-3, Department of Computer Science, Oregon State University, 1990.

[KME89] Kapelnikov (Alex), Muntz (Richard R.) et Ercegovac (Miloš D.). – A modeling methodology for the analysis of concurrent systems and computations. *Journal of Parallel and Distributed Computing*, vol. 6, ń 3, juin 1989, pp. 568–597.

[Knu73] Knuth (Donald E.). – *The art of computer programming*. – Reading, Addison-Wesley, 1973, deuxième édition, *Computer Science and Information Processing*.

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

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

[Lar89] Larousse, Paris. – *Dictionnaire de la langue française – lexis*, 1989.

[LER92] Lewis (Ted G.) et El-Rewini (Hesham). – *Introduction to Parallel Computing*. – Englewood Cliffs, Prentice-Hall International, 1992.

[Lim88] Limited (INMOS). – *Occam 2 reference manual*. – New York, Prentice-Hall International, 1988, *Prentice-Hall International Series in Computer Science*.

[Lim92] Limited (INMOS). – *The Transputer Databook*. – Italy, INMOS Limited – SGS Thomson Microelectronics, 1992. Troisième édition.

[LJ92] Li (Juan) et Jamieson (Leah H.). – Algorithm-architecture mapping using hypergraphs: theory and experimental results. *In : Proceedings of the 1992 International Conference on Parallel Processing*. The Pennsylvania State University, pp. I/210–I/219. – CRC Press.

[LK90] Lee (Pei-Zong) et Kedem (Zvi Meir). – Mapping nested loop algorithms into multidimensional systolic arrays. *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, nº 1, janvier 1990, pp. 64–76.

[LM90] Le Maire (Claude). – *Le langage SIGNAL : un exemple en segmentation automatique de la parole continue*. – Publication Interne nº 527, Rennes, France, IRISA - Institut de Recherche en Informatique et Systèmes Aléatoires, 1990.

[LS77] Lam (Shui) et Sethi (Ravi). – Worst case analysis of two scheduling algorithms. *SIAM J. Comput.*, vol. 6, nº 3, septembre 1977, pp. 518–536.

[MS91] Madal (Sridhar) et Sinclair (James B.). – Performance of synchronous parallel algorithms with regular structures. *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, nº 1, janvier 1991, pp. 105–116.

[MT91] Muntean (Traian) et Talbi (El-Ghazali). – Méthodes de placement statique des processus sur architectures parallèles. *Technique et Science Informatiques*, vol. 10, nº 5, 1991, pp. 355–373.

[MW90] Muntean (Traian) et Waille (Philippe). – L'architecture des machines supernode. *La Lettre du Transputer et des Calculateurs Distribués*, vol. 7, septembre 1990, pp. 11–40.

[Nic89] Nicole (Denis A.). – ESPRIT project 1085: reconfigurable Transputer processor architecture. *In : CONPAR88*, éd. par Jesshope (C. R.) et Reinartz (K. D.). British Computer Society, pp. 81–89. – Cambridge, 1989.

[P+90] Pingali (Keshav) et al. – *Dependence flow graphs: an algebraic approach to program dependencies*. – Rapport Technique nº TR 90-1152, Ithaca, Department of Computer Science, Cornell University, septembre 1990.

[Pla93] Plateau (Brigitte). – *Présentation d'APACHE*. – Rapport Apache 1, Grenoble, IMAG, octobre 1993.

[Pol90] Polychronopoulos (Constantine D.). – *Auto scheduling: control flow and data flow come together*. – Rapport Technique nº CSRD 1058, Urbana, Center for Supercomputing Research and Development, décembre 1990.

[Pop90] Poplawski (D. A.). – *Synthetic models of distributed memory parallel programs*. – Rapport Technique nº ORNL/TM – 11634, ORNL – Oak Ridge, Tennessee 37831 – USA, Oak Ridge National Laboratory – Martin Marietta, 1990.

[PPJ94] Pouzet (P.), Paris (J.) et Jorrand (V.). – Parallel application design: the simulation approach with HASTE. *In : Proceedings of HPCN'94*, éd. par Gentzsch (Wolfgang) et Harms (Uwe). pp. 379–393. – München, BRD, 1994.

[RCJ72] Ramamoorthy (C. V.), Chandy (K. M.) et Jr. (Mario J. Gonzales). – Optimal scheduling strategies in a multiprocessor system. *IEEE Transactions on Computers*, vol. C-21, nº 2, février 1972, pp. 137–146.

[Rob79] Robinson (J. T.). – Some analysis techniques for asynchronous multiprocessor algorithms. *IEEE Transactions on Software Engineering*, vol. SE-5, nº 1, janvier 1979, pp. 24–30.

[Sap90] Saporta (Gilbert). – *Probabilités, analyse des données et statistique*. – Paris, Editions Technip, 1990.

[Sar89] Sarkar (Vivek). – *Partitioning and scheduling parallel programs for multiprocessors*. – London et Cambridge, Pitman et MIT Press, 1989.

[SB78] Stone (Harold S.) et Bokhari (Shahid H.). – Control of distributed processes. *Computer*, vol. 11, nº 7, juillet 1978, pp. 97–106.

[SC92] Scheiman (Chris J.) et Cappello (Peter R.). – A processor-time-minimal systolic array for transitive closure. *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, nº 3, mai 1992, pp. 257–269.

[SGDM94] Sunderam (V. S.), Geist (G. A.), Dongarra (J.) et Manchek (R.). – The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Computing*, vol. 20, nº 4, avril 1994, pp. 531–546.

[SH86] Sarkar (Vivek) et Henessy (John). – Compile-time partitioning and scheduling of parallel programs. *ACM Sigplan Notices*, vol. 21, nº 7, juillet 1986, pp. 17–26.

[T+92] Trystram (Denis) et al. – Ecole Rumeur - communication dans les réseaux de processeurs. – Institut d'Etudes Scientifiques de Cargese (Corse), 1992. Ecole d'été - soutien C3, IMAG, CIMI.

[Tow86] Towsley (Don). – Allocating programs containing branches and loops within a multiple processor system. *IEEE Transactions on Software Engineering*, vol. SE-12, ﻥ 10, octobre 1986, pp. 1019–1024.

[TP90] Touzene (Abderezak) et Plateau (Brigitte). – Mesures de performance des communications du Meganode à 128 Transputers. *La Lettre du Transputer et des Calculateurs Distribués*, vol. 7, septembre 1990, pp. 73–77.

[TP94] Tron (Cécile) et Plateau (Brigitte). – Modelling of communication contention in multiprocessors. *In : Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation - Vienna, Austria,* éd. par Haring (Günter) et Kotsis (Gabriele). pp. 406–424. – Berlin, mai 1994. Lecture Notes in Computer Science 794.

[Ull75] Ullman (J. D.). – NP-complete scheduling problems. *Journal of Computer and System Sciences*, vol. 10, ﻥ 3, juin 1975, pp. 384–393.

[WB93] Waser (Stephan) et Burkhart (Helmar). – OLGA - a modelling tool for algorithmic skeletons. *In : Transputer Applications and Systems '93*, éd. par Grebe (Reinhard) et al. pp. 395–409. – Amsterdam, 1993.

[WG89] Wu (Min-You) et Gajsky (Daniel D.). – Computer-aided programming for message-passing systems: problems and solutions. *Proceedings of the IEEE*, vol. 77, ﻥ 12, décembre 1989, pp. 1983–1991.

[WH94] Wabnig (H.) et Haring (G.). – PAPS - the parallel program performance prediction toolset. *In : Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation - Vienna, Austria*, éd. par Haring (Günter) et Kotsis (Gabriele). pp. 284–304. – Berlin, mai 1994. Lecture Notes in Computer Science 794.

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