Athapascan-1 : interface générique pour l'ordonnancement dans un environnement d'exécution parallèle

Gerson Geraldo Homrich Cavalheiro

▶ **To cite this version:**

Gerson Geraldo Homrich Cavalheiro. Athapascan-1 : interface générique pour l'ordonnancement dans un environnement d'exécution parallèle. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1999. Français. NNT : . tel-00004816

HAL Id: tel-00004816
https://theses.hal.science/tel-00004816v1

Submitted on 18 Feb 2004

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

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

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

T H E S E

pour obtenir le titre de

DOCTEUR DE L'INPG

Spécialité : "INFORMATIQUE : SYSTEMES ET COMMUNICATIONS"

présentée et soutenue publiquement

par

Gerson G. H. Cavalheiro

le 22 Novembre 1999

ATHAPASCAN-1 : Interface générique pour l'ordonnancement dans un environnement d'exécution parallèle

Directeur de thèse :

Jean-Louis Roch
Brigitte Plateau

Composition du jury :

Rapporteurs :	M. Bertil Folliot
	M. Jean-Marc Geib
Examinateurs :	M. Bertil Folliot
	M. Jean-François Méhaut
	Mme Brigitte Plateau
	M. Michel Riveill, *Président*
	M. Jean-Louis Roch

préparée au **Laboratoire de Modélisation et Calcul**
et soutenue au **Laboratoire Informatique et Distribution**
dans le cadre de l'**Ecole Doctorale "Mathématiques et Informatique"**

Remerciements

Le voici le fruit de 4 ans de travail : ma thèse. Bien qu'elle porte seulement mon nom, ce document représente l'engagement d'un grand nombre de personnes. J'essaye ici de ne pas oublier de remercier toutes ces personnes qui m'ont aidé d'une façon ou d'une autre.

Tout d'abord il faut souligner qui cette thèse n'aurait pas vu le jour sans la bourse qui m'a été conférée par la CAPES, dans un accord avec le COFECUB. Merci donc à Marta Elias qui était toujours disponible pour m'assister dans les différentes situations auxquelles je me suis trouvé confronté.

Je tiens à remercier Michel Riveill pour avoir accepté de présider le jury de ma soutenance et Jean-Marc Geib pour la confiance qu'il a accordée à mon travail. Je remercie aussi tout particulièrement Bertil Folliot et Jean-François Méhaut, ils ont pris le temps de lire ce document et de me donner leurs commentaires qui ont été très importants pour sa qualité finale.

Et bien sûr je remercie Brigitte Plateau, qui m'a donné l'honneur de travailler dans son équipe et qui à diverses reprises m'a soutenu lors de mes coups de fatigue...

Enfin un grand merci à mes collèges qui se sont volontiers jetés sur des chapitres de ma thèse pour faire des corrections dans le texte. Comme je pense que quand même j'ai laissé traîner quelques fautes, je ne vais pas leur faire la honte de citer leurs noms dans ce paragraphe.

Mais il n'y a pas eu que la thèse : il y a eu aussi tout un travail qui est venu avant.

Travailler avec Jean-Louis Roch m'a beaucoup appris ; je suis loin de pouvoir oublier mon passage chez les APACHEs sous sa direction. Et les A1-*Boys* avec qui j'ai eu grand plaisir à travailler pendant ces années : Jean-Guillaume, Nicolas et surtout François et Mathias, de vrais guerriers avec lesquels je suis très fier d'avoir fait un bout de chemin. Il y a eu aussi Emmanuel Jeannot et Yves, qui m'ont aidé à mieux digérer certaines parties de ma recherche.

Et je n'oublie pas l'A0-*Team*, Jacques Briat, Marcelo et Alexandre, ce dernier a été le témoin de tout le chemin qui j'ai parcouru jusqu'aux dernières nuits. Sans compter Benhur, le SOS dans les moments du «comment peut-on faire...»

Et il y avait aussi les autres habitants de ce laboratoire : Chassin, Jean-Marc, Gregory, Olivier, Thierry, Gilles, Renaud, Martha, Ekbel, Naddef, Trystram et Briat, avec eux j'ai eu la chance de discuter dans les couloirs, dans la cafétte ou au cours des séminaires – de sujets techniques ou bien d'autres. Et, bien sûr, ma joie ne serait pas complète sans mes copains de bureaux, Alfredo et Christophe : ensemble nous

avons essayé de trouver les réponses aux problèmes les plus fondamentaux de la vie, tels que définir l'horaire d'ouverture des fenêtres...

Un grand merci aussi aux autres brésiliens du laboratoire : Andréa, Gustavo, Paulo et Ricardo, ils ont été en quelque sorte ma famille d'emprunt.

Et je ne peux pas oublier de remercier le soutien très technique de Claudine, Corinne, Helene, Joëlle, Khadija et Philippe.

Finalement, une fois arrivé à la fin, ma pensée revient à ma famille et à ma belle-famille que nous avons laissées au Brésil. La confiance qu'ils m'ont toujours accordée a été d'une grande valeur pour aller jusqu'au bout.

À mon père, qui m'a donné mon premier livre.

Et surtout, surtout, à Luciana, qui a eu les plus divers rôles dans ces 4 ans – d'infirmière à professeur de langue – pendant que je jouais l'étudiant (parfois maladroit...). Si j'ai réussi à remplir le contrat, réel ou moral, avec toute les personnes qui j'ai citées ci-dessus, je le dois avant tout à elle.

Table des matières

1 Introduction **15**

I Ordonnancement dans les environnements de programmation parallèle : état de l'art **19**

2 L'ordonnancement d'une application parallèle **21**

2.1 Définition du problème de l'ordonnancement 22
2.1.1 Caractérisation informelle de l'ordonnancement 22
2.1.2 Visions de l'ordonnancement 23
2.1.2.1 L'ordonnancement côté système 23
2.1.2.2 L'ordonnancement côté application 24
2.2 Comment réaliser l'ordonnancement applicatif 25
2.2.1 Contrôle de la localisation de données 25
2.2.2 Contrôle de la localisation et de l'entrelacement des tâches . 26
2.3 Comment calculer l'ordonnancement applicatif 28
2.3.1 Le sous-système de manipulation d'information 28
2.3.2 Le sous-système de décision 30
2.3.3 Classification des algorithmes d'ordonnancement 31
2.3.4 Grandeurs caractéristiques des applications 32
2.4 Vers une interface générique 33
2.5 Conclusion 35

3 Environnements pour l'ordonnancement **37**

3.1 Équilibrage de charge dynamique 37

3.1.1 GTLB 38
3.1.2 DPC++ 39
3.2 Partage de charge dynamique 40
3.2.1 Cid 41
3.2.2 Cilk 42
3.2.3 Millipede 44
3.2.4 Jade 46
3.2.5 Dynamo 47
3.3 Ordonnancement statique 48
3.3.1 PYRROS/RAPID 49
3.3.2 Metis et SCOTCH 50
3.4 Conclusion 51

II Une interface générique pour l'ordonnancement 55

Avant propos 57

4 Fondements des stratégies d'ordonnancement applicatif 59

4.1 Introduction 60
4.2 Les stratégies statiques 63
4.2.1 Ordonnancements de type partage de charge 63
4.2.1.1 Durée des tâches : ordonnancement de Coffman-Graham 63
4.2.1.2 Prise en compte des communications : ordonnancement ETF 64
4.2.1.3 Regroupement des communications : ordonnancement DSC 64
4.2.2 Ordonnancements de type équilibrage de charge 65
4.2.3 Ordonnancement statique et applications régulières 66
4.3 Les stratégies dynamiques 66
4.3.1 Équilibrage de charge dynamique 67
4.3.2 Partage de charge dynamique 68
4.3.3 Limites théoriques des stratégies dynamiques 69

4.3.3.1 Préemption et performances 70
4.3.3.2 Inefficacité en espace mémoire des stratégies aveugles 71
4.4 Ordonnancement hybride 72
4.5 Fondements communs : l'analyse du flot de données 73
4.5.1 Représentation de l'exécution par un graphe 74
4.5.2 Formalisation du problème d'ordonnancement 77
4.6 Le constructeur de tâches : l'interface entre l'application et l'ordonnancement 79
4.7 Conclusion 80

5 L'exécutif : interface entre l'ordonnancement et l'architecture 83
5.1 Outils de base pour la programmation des architectures parallèles . . 83
5.1.1 Différents types d'architecture parallèle 84
5.1.2 Exploitation du parallélisme inter-nœuds et communications 85
5.1.3 Recouvrement des temps d'attente par le *multithreading* . . 85
5.2 Une abstraction de machine parallèle 86
5.3 Mécanismes pour l'ordonnancement applicatif 87
5.3.1 Manipulation des données 87
5.3.2 Entrelacement de tâches 88
5.4 L'exécutif : l'interface entre l'ordonnancement et l'architecture . . . 88
5.4.1 L'interface de l'exécutif 89
5.5 Conclusion 90

6 Proposition d'un noyau générique pour l'ordonnancement 93
6.1 Introduction 93
6.2 L'interface du niveau d'ordonnancement 94
6.2.1 L'ordonnancement local sur une architecture SMP 94
6.2.2 L'ordonnancement global sur une architecture distribuée . . 96
6.3 Implantation du noyau d'ordonnancement 98
6.3.1 Le sous-système de décision 99
6.3.2 Le sous-système d'inspection de charge 100
6.3.3 Contrôle centralisé *vs.* contrôle réparti 101

6.4 Implantation par classes 102
6.5 Validation de l'environnement générique 103
6.6 Conclusion 104

III Ordonnancement pour ATHAPASCAN 105

Avant propos 107

7 Integration de l'ordonnancement au sein d'ATHAPASCAN 109

7.1 ATHAPASCAN-1 et l'ordonnancement 110
7.1.1 ATHAPASCAN-1 : l'interface applicative d'ATHAPASCAN . 110
7.1.2 L'interface avec l'ordonnancement 111
7.1.3 L'implantation 113
7.1.4 Utilisation de l'ordonnancement applicatif 114
7.2 ATHAPASCAN-0 et l'ordonnancement 115
7.2.1 ATHAPASCAN-0 : le noyau exécutif d'ATHAPASCAN 116
7.2.2 L'interface avec l'ordonnancement 117
7.2.3 ATHAPASCAN-0 comme support à l'ordonnancement 118
7.2.3.1 Le pool d'exécution 118
7.2.3.2 Les communications 121
7.3 Conclusion 121

8 Mise en œuvre des stratégies d'ordonnancement 123

8.1 L'ordonnancement applicatif 123
8.1.1 Le sous-système de décision 124
8.1.1.1 Implantation d'une politique 124
8.1.1.2 Initialisation du noyau d'ordonnancement 126
8.1.1.3 Exécution des opérations d'ordonnancement . . . 127
8.1.2 L'inspecteur de charge 127
8.1.2.1 Configuration de l'inspection de charge 127
8.1.2.2 Exécution des opérations de l'inspection de charge 128
8.2 Cohabitation des politiques d'ordonnancement 129
8.2.1 Le problème 129

8.2.2 Solution retenue 131
8.3 Utilisation pratique 132
8.3.1 Un algorithme d'équilibrage de charge 133
8.3.2 Un algorithme de liste 134
8.3.3 Un algorithme basé sur le vol de travail 135
8.3.4 Un algorithme statique 136
8.3.5 La spécialisation des stratégies 137
8.4 Conclusion 137

9 Évaluation du noyau d'ordonnancement 139
9.1 Architectures utilisées et conditions d'expérimentation 140
9.2 Une application récursive 140
9.3 Une Application numérique 142
9.4 Une stratégie mixte 144
9.5 Les surcoûts du noyau d'ordonnancement 146
9.5.1 Identification des surcoûts 147
9.5.2 Influence de l'ordonnancement dans le surcoût d'exécution d'un programme ATHAPASCAN-1 150
9.6 Compression de fichiers 151
9.7 Visualisation d'une politique d'ordonnancement 153
9.8 Bilan des résultats 153

10 Conclusion 157

A La classe A1_DECISION 161

B L'ordonnancement dans un programme ATHAPASCAN-1 163

Table des figures

1.1 L'environnement d'ordonnancement générique proposé 16
1.2 Environnement de programmation parallèle d'ATHAPASCAN 17

2.1 Les composants d'un environnement d'exécution parallèle 22
2.2 Identification des interfaces entre les modules d'un environnement d'exécution parallèle 34

4.1 Un graphe de précédence 76
4.2 Un graphe de flot de données 76
4.3 Le constructeur de tâches : l'interface entre l'application et l'ordonnancement 79

5.1 Représentation d'une machine parallèle abstraite 87
5.2 L'exécutif : l'interface d'accès aux ressources de la machine parallèle 89

6.1 Un environnement d'exécution parallèle doté d'un noyau d'ordonnancement générique 94
6.2 Interface du noyau d'ordonnancement dans une architecture SMP . 95
6.3 Interface du noyau d'ordonnancement dans une architecture distribuée 97
6.4 Structure interne du niveau d'ordonnancement et les interactions entre les réplicats 99

7.1 Un exemple de programme ATHAPASCAN-1. 110
7.2 Services requis et fournis par le niveau d'ordonnancement 112
7.3 Un instant dans l'évolution du graphe de flot de données 114

8.1 Le graphe de flot de données d'une application parallèle de type Jacobi 130
8.2 Gestion de multiples politiques d'ordonnancement 132

9.1 Performance d'un algorithme de factorisation de Cholesky sur 16 nœuds d'un IBM-SP 143
9.2 Performance d'un algorithme de factorisation de Cholesky sur un noyau d'ordonnancement spécialisé 144
9.3 Performance d'un algorithme de compression de fichiers 152
9.4 Visualisation de l'exécution du noyau d'ordonnancement 154

Liste des tableaux

3.1 Modèles de programmation des environnements étudiés 51
3.2 Récapitulatif des caractéristiques d'ordonnancement des environnements étudiés 53

4.1 États des nœuds d'un graphe de flot de données. 78

7.1 Temps liées aux opérations sur les processus légers et aux manipulations de verrous. 120

9.1 Le problème des n-Reines sur une architecture parallèle 141
9.2 Le problème des n-Reines sur une architecture SMP et un NOW hétérogène 141
9.3 Utilisation d'une stratégie d'ordonnancement mixte 145
9.4 Influence du nombre de flots d'exécution concurrents sur différentes architectures 147
9.5 Opérations réalisées par un algorithme glouton 149
9.6 Surcoûts d'une exécution distribuée 150
9.7 Réduction du surcoût de manipulation du graphe par le vol de travail 151

1

Introduction

Il y a déjà plus de deux décennies que le calcul parallèle est utilisé dans de nombreux domaines applicatifs [12, 109], comme [7, 120, 43], Néanmoins, les interfaces de programmation parallèle disponibles sur de nombreuses architectures aujourd'hui (RPC, *sockets*, PVM [62], MPI [110], POSIX [1], *etc*) offrent un faible confort de programmation. Notamment, contrairement à la programmation séquentielle, un point important est le fait que ce type d'interface ne permet pas de s'abstraire des caractéristiques de l'architecture sous-jacente (par exemple le nombre de processeurs). Ceci limite évidemment la portabilité des programmes.

Ainsi, différents travaux de recherche sont motivés par la conception de nouvelles interfaces de programmation parallèle qui offrent un compromis entre confort de programmation, portabilité et efficacité. Parmi ceux-là, on peut notamment citer Cilk [14], Jade [102] et ATHAPASCAN-1 [58]. La portabilité est garantie par une sémantique indépendante de l'architecture. Le contrôle de cette sémantique est généralement réalisé lors de l'exécution, par un mécanisme d'ordonnancement (routage des données, entrelacement des calculs) sur l'architecture cible. Pour des programmes possédant un grand degré de parallélisme, de nombreux choix d'ordonnancement sont possibles qui tous respectent la sémantique de l'interface. Les performances d'un programme sur une architecture étant directement dépendantes de l'ordonnancement, le noyau qui le réalise est alors un élément critique au sein de l'environnement.

En pratique, le choix de la stratégie d'ordonnancement la plus performante dépend à la fois de l'application [105, 60] et de l'architecture cible. Dès lors, un compromis possible entre portabilité et performance est de permettre la modification ou

l'ajustement de la stratégie de régulation par annotation du code. Le problème qui se pose est alors de dissocier le code applicatif de la spécification de son ordonnancement sur l'architecture cible.

Objectifs de la thèse

Dans cette thèse, nous proposons la description dynamique du flot de donnée associé à une exécution comme l'élément central permettant de séparer le code applicatif de la régulation sur une architecture cible. Nous basons cette proposition sur la spécification des interfaces de la régulation avec l'application d'une part et avec l'architecture d'autre part (cf. figure 1.1). Cette spécification repose sur une identification du problème de la régulation.

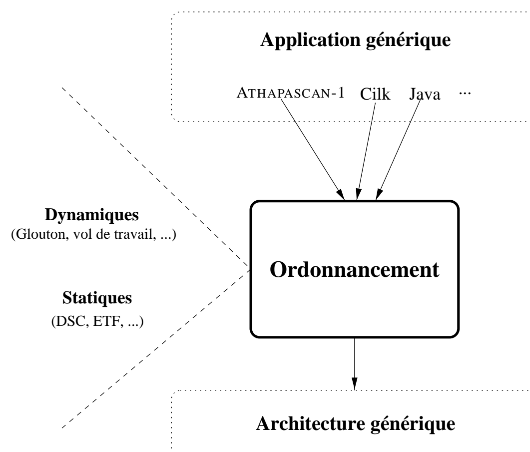

Figure 1.1 *L'environnement d'ordonnancement générique proposé.*

Nous justifions la pertinence de cette spécification en montrant :

– qu'elle peut être interfacée (avec éventuellement certaines restrictions) pour différentes interfaces de programmation parallèle, notamment Cilk, Jade et ATHAPASCAN-1.
– qu'elle peut être implantée sur différentes architectures : séquentielle, distribuée, SMP.
– qu'elle permet d'implanter de très nombreuses et variées stratégies de régulation : statiques, dynamiques et hybrides.

Cadre de travail

Le noyau générique d'ordonnancement que nous proposons dans cette thèse a été intégré au sein de l'interface de programmation parallèle ATHAPASCAN-1 du projet APACHE, Algorithmique Parallèle et pArtage de CHargE[1] [99]. La spécification proposée a joué un rôle moteur dans la définition de cette interface. Elle a été implantée sur le noyau exécutif portable pour architectures parallèles, ATHAPASCAN-0 du projet APACHE.

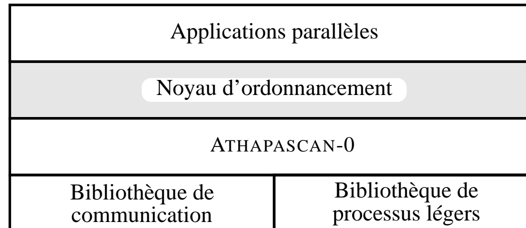

Figure 1.2 *L'environnement de programmation parallèle* ATHAPASCAN *est composé de diverses couches. Cette thèse est centrée sur le noyau d'ordonnancement qui permet l'exécution de programmes écrits en* ATHAPASCAN-1 *sur le noyau exécutif* ATHAPASCAN-0.

ATHAPASCAN-0. ATHAPASCAN-0 [20, 64, 22] est une couche de programmation parallèle de bas niveau capable de créer un réseau dynamique de processus légers communiquant. ATHAPASCAN-0 réunit dans un seul outil des mécanismes élémentaires de communication entre processus et de création des processus légers. ATHAPASCAN-0 est bâti sur des bibliothèques standards de communication et de processus légers.

ATHAPASCAN-1. ATHAPASCAN-1 [58, 43] est une interface de programmation de haut niveau permettant la description d'un programme parallèle par la création de tâches. Un graphe de flot de données dynamique décrivant l'exécution du programme est créé à partir des accès des tâches à des données partagées [59].

[1] Projet joint CNRS–INPG–UJF–INRIA (Centre National de la Recherche Scientifique – Institut National Polytechnique de Grenoble – Université Joseph Fourier – Institut National de la Recherche en Informatique et en Automatique.
Adresse WEB : `http://www-apache.imag.fr`.

Organisation de ce document

Ce document est structuré en trois parties.

La première partie est consacrée à un état de l'art sur les techniques de régulation de charge (chapitre 2) et leur utilisation dans différents environnements de programmation parallèle (chapitre 3). Dans cette partie, nous proposons une formalisation du problème de régulation qui sert de base à l'identification des interfaces de la régulation avec l'application d'une part et l'architecture d'autre part.

La deuxième partie est consacrée à la spécification de ces deux interfaces (chapitres 4 et 5) ; le chapitre 4 qui propose l'analyse du flot de données comme interface entre l'application et la régulation joue un rôle central. Dans le chapitre 6 nous proposons une librairie générique (noyau d'ordonnancement) permettant l'implantation de différentes stratégies de régulation statiques ou dynamiques.

Enfin la troisième partie de cette thèse présente une implantation effective pour ATHAPASCAN-1 des interfaces abstraites proposées dans la deuxième partie (chapitre 7). Nous détaillons l'implantation de quelques stratégies (chapitre 8) et nous analysons leurs performances sur quelques applications didactiques (chapitre 9).

Première partie

Ordonnancement dans les environnements de programmation parallèle : état de l'art

2

L'ordonnancement d'une application parallèle

Au sein d'un environnement d'exécution parallèle le contrôle de la répartition et de l'exécution d'une application sur une architecture est délégué à un noyau d'ordonnancement. Dans ce chapitre nous abordons le probléme lié à ce contrôle pour identifier ses caractéristiques dans un tel environnement d'exécution.

Dans un premier temps, section 2.1, nous définissons d'une manière informelle l'ordonnancement d'une application parallèle. Nous montrons que cet ordonnancement peut être appliqué à deux niveaux : l'ordonnancement système et l'ordonnancement applicatif.

Ensuite, dans la section 2.2, nous concentrons notre étude sur l'ordonnancement applicatif, c'est-à-dire sur l'ordonnancement de programmes parallèles. Nous présentons les mécanismes employés par le processus d'ordonnancement : essentiellement ceux destinés à la manipulation de tâches et de données. Nous présentons aussi une taxonomie des algorithmes d'ordonnancement.

Dans la section 2.3 sont présentés les différents composants des algorithmes d'ordonnancement : les sous-systèmes de décision et d'inspection de charge.

Finalement, dans la section 2.4, nous caractérisons l'ordonnancement comme un module indépendant de l'application et de l'architecture. Nous identifions la nécessité d'une interface de services entre eux.

2.1 Définition du problème de l'ordonnancement

Nous voulons dans cette première section définir la vision du problème de l'ordonnancement que nous développons dans cette thèse. La section 2.1.1 présente la notion d'ordonnancement face à une application et à une machine. Dans la section 2.1.2 deux niveaux d'ordonnancement sont identifiés : le niveau système et le niveau applicatif.

2.1.1 Caractérisation informelle de l'ordonnancement

Pour étudier le problème de l'ordonnancement, une première distinction doit être faite entre la formalisation de la structure d'un programme parallèle et l'exploitation de cette structure par un mécanisme de support à l'exécution [37, 125]. Alors que la structure d'un programme parallèle décrit un ensemble de processus et leur relations (synchronisation), le mécanisme de support d'exécution vise l'exploitation des ressources de l'architecture pour les exécuter : ce mécanisme est implanté par un noyau d'ordonnancement.

Le rôle d'un tel noyau d'ordonnancement est d'attribuer à chaque processus généré par une application, l'ensemble des ressources nécessaires à son déroulement à partir d'une date de début et de veiller à ce que tous les processus terminent [124]. En particulier nous considérons que cet ordonnancement supporte l'exécution d'une application parallèle sur une architecture multiprocesseur.

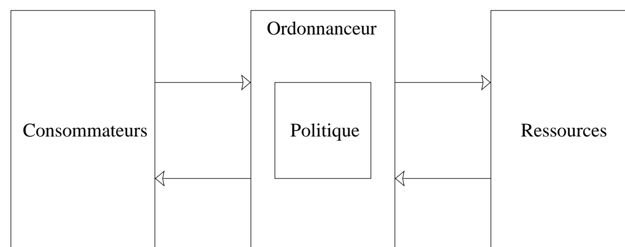

Figure 2.1 *Les composants d'un environnement d'exécution parallèle selon Casavant et Kuhl [24]. Une couche d'ordonnancement est placée entre l'application (le consommateur) et l'architecture (les ressources). L'ordonnancement est responsable de la gestion des ressources disponibles selon les besoins du consommateur.*

Cet ordonnancement, une fois implanté, constitue une couche logicielle placée entre l'application et l'architecture qui effectue la gestion des *ressources matérielles*. Les consommateurs, ou clients, de cette couche sont les applications et les ressources matérielles sont les machines parallèles [24]. Cette structure est présentée par la figure 2.1. Il est possible de remarquer que lors de son implantation l'algorithme d'ordonnancement est dépendant de la nature de l'application [38, 105] et de celle de la machine parallèle [66, 87].

2.1.2 Visions de l'ordonnancement

L'ordonnancement dans un environnement d'exécution parallèle est normalement supporté par deux niveaux [47] : le niveau d'ordonnancement *système* et le niveau d'ordonnancement *application*. Le premier est dédié à allouer les ressources de calcul nécessaires au programme ; le deuxième est le responsable de l'attribution des éléments de l'application (processus et synchronisation) aux ressources.

Cette division à deux niveaux permet d'une part d'alléger la tâche de l'ordonnancement système et d'autre part d'avoir des schémas d'ordonnancement optimisés en fonction de l'application.

2.1.2.1 L'ordonnancement côté système

Nous appelons l'ordonnancement système le niveau d'ordonnancement responsable de la gestion des programmes utilisateur dans une architecture multiprogramée [47]. Cet ordonnancement est généralement supporté par un système d'exploitation et implique le problème de l'allocation des ressources, c'est-à-dire de décider où et quand exécuter chaque processus. En outre, il alloue les espaces mémoire pour les données et les espaces disques aux fichiers. Il est aussi responsable en partie de l'acheminement des messages transmis entre les procesus.

L'ordonnancement de processus implique un partage de processeurs de la machine. Ce partage peut être temporel et spatial.

Le partage temporel définit le mécanisme à employer pour entrelacer dans le temps l'accès des processus aux processeurs. La technique peut-être la plus utilisée est celle de maintenir pour chaque processeur une liste des processus actifs et d'attribuer à chaque processus de cette liste des quotas de temps. Eventuellement cette liste peut être maintenue selon un principe quelconque de priorité. Nous retrouvons ce mécanisme de partage du temps dans les systèmes Unix compatibles ou dans des environnements construits sur Unix, par exemple Mach [9] ou sur la norme POSIX [1].

Le partage spatial définit l'attribution des processus aux processeurs de la ma-

chine ; il détermine quel processeur exécutera quels processus. Ce partage permet l'exécution concurrente de diverses applications. Nous retrouvons le partitionnement d'une machine parallèle en groupes de processeurs, en attribuant chaque groupe à une application [47]. D'autres outils intégrent des mécanismes permettant l'obtention de performances, par exemple en terme de temps d'exécution des applications [107, 53].

2.1.2.2 L'ordonnancement côté application

Du côté application, l'ordonnancement permet de réaliser l'exécution d'un programme parallèle sur une architecture multiprocesseur. De manière assez générale, l'exécution d'un programme parallèle peut être représentée par un graphe dans lequel sont indiquées les relations de dépendance entre les instructions [105] ; nous allons employer le mot *tâche* pour décrire une suite séquentielle d'instructions et le mot *synchronisation* pour décrire une communication entre deux tâches (une relation de dépendance)[1].

L'ordonnancement de niveau applicatif associe à chaque tâche un processeur pour son exécution et une date de début. Nous retrouvons dans [105] l'identification de trois grandes classes d'applications parallèles en fonction du graphe qui les représente :

1. Applications régulières, générant un graphe *prévisible*. Cette classe est composée d'applications pour lesquelles il est possible de déterminer le graphe avant le démarrage de l'exécution en analysant les données d'entrée.
2. Applications irrégulières, générant un graphe *imprévisible*. Le graphe est entièrement créé lors de l'exécution du programme.
3. Applications semi-régulières, générant un graphe *semi-prévisible*. Le graphe est structuré en une succession de phases, chaque phase décrivant un sous-graphe qui dépend des données en sortie de la phase précédente ; ainsi, pour chaque sous-graphe, il est possible d'appliquer les techniques pour les graphes prévisibles.

L'ordonnancement applicatif est souvant caractérisé par le type de graphe généré par l'application. Nous retrouvons dans la littérature des environnements d'ordonnancement applicatif qui exploitent explicitement un programme parallèle décrit par un graphe. C'est le cas de PYRROS [130], Cilk [14] et Jade [101]. Tandis que PYRROS est appliqué sur un graphe entièrement décrit, Cilk et Jade appliquent l'ordonnancement sur un graphe créé lors de l'exécution d'un programme.

[1]Nous allons raffiner ces définitions dans le chapitre 4 lors de la discussion sur la représentation d'un programme parallèle par un graphe de flot de données.

Dans la suite de ce document nous abordons ce niveau d'ordonnancement.

2.2 Comment réaliser l'ordonnancement applicatif

Dans cette section nous présentons les opérations qui sont effectuées lors de l'ordonnancement pour contrôler l'exécution d'un programme parallèle. Ces opérations permettent de contrôler la distribution des données dans les modules mémoire et le placement et l'entrelacement des tâches sur une architecture multiprocesseur à mémoire distribuée [19, 60].

2.2.1 Contrôle de la localisation de données

Pour contrôler la distribution et l'accès des données parmi les modules de mémoire disponibles sur une architecuture distribuée il est nécessaire la présence d'un réseau de communications. Nous considérons que des primitives permettant l'envoi et la réception de messages sont disponibles et que l'ordonnancement système se charge de l'acheminement des messages entre un processeur source et un processeur destinataire[2].

L'ordonnancement applicatif gère alors le placement et la migration de données sur les modules de mémoire de la machine parallèle. Le placement consiste à attribuer à une donnée un espace mémoire sur un des modules de mémoire. La migration consiste à enlever une donnée placée sur un module pour la placer sur un autre. Ces opérations sont réalisées à partir des échanges de messages.

De manière à réduire le surcoût dû aux échanges de données, différentes techniques on été développées, notament la réduction du nombre de messages et le recouvrement des communications par les calculs.

Pour réduire le nombre de messages échangés lors de l'exécution d'un programme, des techniques cherchent à augmenter la localité des données, c'est-à-dire de faire en sorte que les données en mémoire distribuée soient le plus possible accédées localement. Une méthode très répandue est la règle *owner compute rule*, qui consiste à placer un calcul sur le site où se trouvent les données sur lesquelles il travaille. Le principe consiste à distribuer sur les mémoires locales de chaque processeur les données manipulées par le programme et à exécuter chaque tâche sur le processeur qui contient les données dont elle a besoin [97]. Éventuellement, de

[2]Pour cela, les données sont découpées en paquets et communiquées selon une politique spécifique [37].

manière à assurer une bonne répartition de la charge, il peut être nécessaire de réaliser une redistribution des données en cours d'exécution [50]. D'autres méthodes utilisent des mécanismes plus élaborés pour la gestion des données, prévoyant, par exemple, leur duplication [36]. Une autre technique complementaire pour réduire le nombre des messages consiste à regrouper sur un même processeur des activités communicantes [78, 130].

Dans le cas où il n'est pas possible d'éviter les communications, il est toujours possible de masquer leur surcoût en évitant que le processeur soit inactif pendant les échanges de messages. Deux techniques peuvent être utilisées, soit prévoir les échanges futurs et les anticiper en utilisant des primitives asynchrones de communication à attente différée, soit permettre aux processeurs d'avoir des flots multiples d'exécution (*multithreading*). Ainsi, dès qu'un flot d'exécution entre dans une phase d'échange de données, un autre flot peut occuper les ressources de calcul du processeur [121].

En simulant plusieurs processeurs logiques, ou virtuels, sur un même processeur physique le *multithreading* permet de pipeliner les communications pour en masquer la latence ; si le réseau est complet, ce masquage est total [75]. De nombreux travaux de différents natures utilisent cette technique : théorique ([122] parle de *parallel slackness*, [75] parle de la simulation d'une PRAM sur une architecture distribuée), matériel ([21]) et surtout logiciel de base (Nexus [55], PM^2 [89], ATHAPASCAN-0 [20]). La technique classique [121] consiste à simuler q processeurs virtuels sur p processeurs de l'architecture physique ($q = p \log p$), en utilisant généralement des processus légers (*threads*). La simulation est alors dite optimale si le délai pour un accès est proportionnel à $\frac{q}{p}$.

2.2.2 Contrôle de la localisation et de l'entrelacement des tâches

Le problème du placement et de l'entrelacement de tâches consiste à déterminer quelles tâches s'exécuteront sur un processeur et à quelles dates.

Avant de pouvoir initier le traitement d'une tâche – l'exécuter –, celle-ci doit être attribuée à un processeur ; cette association d'une tâche à un processeur est réalisée par le mécanisme de placement de tâches. L'opération de placement peut être réalisée tant sur des tâches prêtes que sur des tâches non-prêtes – une tâche non-prête est une tâche dont le démarrage est dépendant de la terminaison d'autres tâches.

Une tâche prête est considérée activable dès qu'elle est placée. Elle peut alors être démarrée. La tâche devient alors en exécution. Il est possible de distinguer deux stratégies employées par les mécanismes de démarrage :

1 *démarrage automatique*, dans lequel toute tâche qui devient activable est automatiquement démarrée [38].

2 *démarrage différé*, dans lequel il est possible de retarder l'exécution des tâches activables [86].

Il est important de noter que, à un instant donné, un processeur ne peut supporter que l'exécution d'une seule tâche, même si plusieurs tâches peuvent être dans l'état exécution. Le partage du temps d'utilisation du processeur est garanti par un mécanisme de « préemption ». On distingue différents types de préemption :

1. *Non-préemptif* ou *préemptif uniquement dans des points de synchronisation* : une tâche démarrée est exécutée jusqu'à sa terminaison ou jusqu'à ce qu'une primitive de synchronisation soit exécutée par cette tâche, par exemple la prise d'un sémaphore ou l'attente d'une communication. Dans ce dernier cas, la tâche devient bloquée en attendant l'événement externe qui la fera retourner à l'état d'exécution.
2. *Préemptif* ou *préemptif avec partage de temps* : chaque tâche en exécution sur un processeur a droit à un quota de temps. Si une tâche n'est pas terminée avant la fin du quota qui lui a été attribué, elle est retirée du processeur pour permettre à une autre tâche d'avancer. Le traitement est réalisé comme dans le cas non-préemptif lorsque la tâche fait appel à une primitive de synchronisation.

Nous observons que les mécanismes employant la préemption permettent de masquer les temps d'attente auxquels les tâches sont exposées lors des appels à des primitives de synchronisation [12, 109, 35], notamment celles impliquées par des primitives de communications [61].

Un autre problème qui se pose est d'associer une bonne répartition des tâches entre les processeurs. Cette repartition est réalisée dans un premier temps par le placement de tâches ; cependant un placement déjà réalisé peut être remis en cause : on fait alors recours à une opération de migration de tâche. La migration est un mécanisme qui permet d'enlever une tâche placée sur un processeur et de la replacer sur un autre. La migration peut considérer les tâches non démarrées ou celles en cours d'exécution :

1. *Migration restreinte*. Ce type de migration prend une tâche qui est déjà affectée à un processeur mais qui n'a pas encore été démarrée. Ce schéma de migration est particulièrement adapté à un noyau d'exécution non-préemptif : dans ce cas, une tâche en exécution ne peut pas être considérée comme candidate à la migration et devra s'exécuter jusqu'au bout.
2. *Migration d'une tâche en exécution*. Dans le cas où les tâches sont communicantes, la mise dans l'état d'exécution ne peut pas être retardée au risque d'une

situation d’interblocage [127]. La migration dans une telle situation requiert alors un mécanisme de préemption de tâches. Le surcoût de la migration est dans ce cas beaucoup plus élevé, puisque toute l’image de l’exécution de la tâche (les registres, la mémoire, *etc*.) doit être transférée d’un processeur à l’autre [89, 38].

Nous avons présenté dans les deux dernières sections les mécanismes qu’un noyau d’ordonnancement peut employer pour manipuler les données et les tâches générées par un programme. Dans la section suivante nous caractérisons le fonctionnement de l’ordonnancement, qui utilise les mécanismes décrits.

2.3 Comment calculer l’ordonnancement applicatif

Nous cherchons à caractériser dans cette section le fonctionnement d’un algorithme d’ordonnancement. Pour cela, nous identifions les composants internes d’un tel algorithme. Nous centrons notre discours sur le problème de l’ordonnancement de tâches ; cependant il reste tout à fait valide pour l’ordonnancement de données.

Un algorithme d’ordonnancement doit être capable de répondre à quelques questions simples, concernant le placement et la migration des tâches pour pouvoir accomplir sa fonction [25, 5]. Les réponses à ces questions sont cherchées dans des politiques qui exploitent les caractéristiques de l’application et de la machine. Nous distinguons ainsi deux sous-systèmes internes aux algorithmes d’ordonnancement : l’un de *décision*, pour manipuler les tâches et contrôler leur exécution sur la machine, et un autre de *manipulation d’information*, pour observer la charge du système [127, 116, 25]. Bien que, à l’implantation, ces deux modules soient étroitement liés, dans une spécification fonctionnelle ils peuvent être analysés séparément [73].

Les politiques considérées dans l’implantation de chacun de ces modules sont présentées dans la suite [127, 53, 25].

2.3.1 Le sous-système de manipulation d’information

Le sous-système de manipulation d’information surveille l’évolution des taux d’utilisation des ressources de l’architecture et fournit au sous-système de décision des indications sur la charge courante de la machine [116, 25]. Ces indications sont plus ou moins précises selon les besoins des politiques de décision. Pour certaines

de ces politiques, dites aveugles [65, 14, 71, 11], aucune information de charge n'est nécessaire.

Ces informations étant observées localement sur chaque processeur de l'architecture, leur manipulalation nécessite la spécification de deux politiques :

1. Politique d'*évaluation de charge d'un processeur*, qui définit la fonction utilisée pour mesurer la charge individuelle sur chaque processeur et la métrique pour la représenter.
2. Politique de *dissémination d'informations*, qui détermine les processeurs concernés par un échange d'information et le moment du démarrage de la réalisation de cet échange.

Divers paramètres peuvent être pris en compte par une fonction d'évaluation [92]. Par exemple, le nombre de tâches en attente d'exécution [127] ou en cours exécution [38]. D'autres algorithmes utilisent encore des mesures représentant les taux d'utilisation de la mémoire des processeurs [83]. Nous trouvons aussi des algorithmes qui prennent un ensemble de paramètres pour fournir un seul indice de charge [49]. Une fois définis le paramètre de l'évaluation, reste à définir la forme de sa représentation : un indice peut être présenté par une valeur précise, par exemple, une valeur numérique [49], ou par une valeur plus abstraite qui donne une pré-évaluation qualitative de l'état du processeur, indiquant s'il est *faiblement*, *normalement* ou *fortement* chargé [115, 37].

Quant à la politique de dissémination d'informations, le nombre de solutions est assez réduit [81]. Les solutions à ce problème sont liées à deux points : (*i*) à la fréquence des échanges [115] – périodiques, réalisés à la demande ou à la suite d'une variation importante de la charge d'un nœud – et (*ii*) à la définition des processeurs participant à une action d'échange d'informations [116, 37]. Par exemple le concept de voisinage dans [127] ou la notion d'anneau[3] introduite dans [38], cf. [52].

D'une manière générale les informations charge qui sont considérées en premier lieu sont celles qui représentent les taux d'utilisation des processeurs. Cependant d'autres indices peuvent être aussi représentatifs[4]. Par exemple :

1. *Utilisation du processeur.* L'indice le plus classique de charge représente les taux d'utilisation des ressources de calcul de la machine. Diverses fonctions peuvent être utilisées pour le calculer, par exemple le nombre d'activités en exécution (GTLB [38]) ou en attente d'exécution (Willebeek-LeMair et Reeves [127]).

[3]Nous discutons ces deux exemples dans la section 4.3.1.

[4]Des renseignements supplémentaires sur les indices de charge sont fournis lors de la présentation des outils d'ordonnancement (chapitre 3). Le lecteur pourra aussi se référer à [48, 49, 131, 54, 67].

2. *Volume de mémoire occupé.* Les processeurs disposent d’un espace mémoire limité (Narlikar [90]). Un indice de charge permet représenter la mémoire disponible dans un instant de temps.
3. *Utilisation du réseau.* Le réseau qui rend possible la communication entre les nœuds possède une capacité limitée de trafic par unité de temps. Le taux de saturation du réseau peut être représenté par la quantité d’octets transmis dans un intervalle de temps récent.
4. *Volume d’E/S (entrée/sortie).* D’une façon semblable au réseau, le nombre d’accès qui peuvent être réalisées sur un disque sont limitées. Le nombre d’accès réalisés et la moyenne de la quantité d’octets transférés vers un disque est alors représentative d’une charge (GatoStar [53]). Toujours en référence aux disques, l’information de l’espace disponible est aussi utile en certains cas.
5. *Interactivité.* L’information de la présence d’un utilisateur sur un nœud de la machine est fréquemment employée dans des environnement d’exécution distribués, notamment sur des NOWs. En considérant les taux d’occupation des différents ressources d’un tel réseau de stations de travail, l’environnement peut ainsi exploiter les ressources de calcul inactives (MARS [68]).

2.3.2 Le sous-système de décision

Dans le sous-système de décision sont définies et implantées les politiques de placement et de migration de tâches [127, 53, 44, 5] :

1. Politique de *déclenchement*, qui déclenche d’une action d’ordonnancement.
2. Politique de *localisation*, qui détermine le processeur approprié pour recevoir une tâche.
3. Politique de *transfert* ou de *sélection*, qui choisit les tâches à transmettre d’un processeur à l’autre.

Typiquement deux solutions sont proposées pour déterminer la nécessité de déclencher une phase d’ordonnancement [127] : une méthode qui prévoit un démarrage périodique [111] et une autre réactive à des événements, par exemple la détection d’une situation de surcharge ou de sous-charge d’un processeur [127] ou la création d’une nouvelle activité au niveau applicatif. La politique de localisation détermine les processeurs concernés par l’opération d’ordonnancement, c’est-à-dire ceux à qui des tâches seront enlevées et ceux qui les recevront. Pour la politique de sélection le problème est de définir quelle est la tâche (ou les tâches) la plus appropriée pour le placement ou la migration.

Nous présentons dans le chapitre 3 quelques algorithmes d’ordonnancement.

2.3.3 Classification des algorithmes d'ordonnancement

Dans cette section nous reprenons la taxonomie de T. Casavant et J. Kuhl [24] pour définir les termes liés à l'ordonnancement que nous utilisons dans cette thèse. Bien que d'autres classifications existent, notamment celles proposées par Bernard [6] et Talbi [117], nous avons choisi de travailler avec celle de Casavant et Kuhl, peut-être la plus souvent citée dans la littérature. Le travail de C. Jacqmot [73] présente une discussion plus approfondie sur cette classification et propose une extension. Nous gardons néanmoins la classification originale qui est basée sur une analyse de la conception des algorithmes d'ordonnancement.

Ordonnancement local et ordonnancement global. Une première distinction entre les politiques d'ordonnancement est basée sur l'identification de leur champs d'action. Ainsi un ordonnancement *local* est responsable de la gestion d'accès des tâches d'un programme en exécution aux ressources d'un processeur. L'attribution d'une tâche à un processeur est du ressort d'un ordonnancement *global*.

Ordonnancement statique et ordonnancement dynamique. Des nombreuses stratégies peuvent être employées par l'ordonnancement global. Selon le moment où elles sont appliquées, une stratégie est dite *statique* ou *dynamique* [84]. Les algorithmes statiques sont applicables pour des programmes dans lesquels il est possible d'obtenir la description de tous les calculs à exécuter avant le démarrage. Une analyse de cette description permet de calculer à l'avance le processeur et la date à laquelle chaque tâche sera exécutée. Dans le cas où les tâches ne sont connues que lors de l'exécution du programme, l'ordonnancement applique des stratégies dynamiques. Ce type d'algorithme prévoit une activation à chaque changement d'état du programme en exécution, par exemple au moment de la création ou de la terminaison d'une tâche ; un surcoût d'ordonnancement est donc ajouté au coût total d'exécution du programme. Bien que les algorithmes dynamiques ne puissent pas analyser la description complète du problème, ils peuvent suivre l'évolution des taux d'utilisation des ressources de la machine par un, voir plusieurs, indices de charge. Dans les stratégies statiques ces indices de charge ne sont pas disponibles, cependant elles peuvent considérer lors de l'ordonnancement une description de l'architecture (vitesse des processeurs, capacité du réseau, *etc.*) [130, 96].

Ordonnancement centralisé et ordonnancement distribué. Selon la nature de la politique d'ordonnancement global appliquée, les décisions sont implantées de façon *centralisée* ou *distribuée*. Dans une politique centralisée un nœud de la machine parallèle prend les décisions de toutes les opérations d'ordonnancement.

Dans une politique *distribuée* au contraire, les opérations d'ordonnancement sont réalisées par tous les nœuds.

Degré de coopération. Le niveau de *coopération* entre les nœuds de la machine permet d'identifier le degré d'indépendance de chaque nœud dans la prise de décision d'une stratégie dynamique distribuée. Un faible degré de coopération indique que chaque nœud prend les décisions d'ordonnancement en tenant compte seulement de sa propre charge, tandis que dans une stratégie coopérative, les décisions prises par un nœud prennent en compte l'impact de la réalisation d'une opération d'ordonnancement sur la charge globale de la machine.

Régulation de charge. Réguler la charge d'un programme implique de distribuer entre les nœuds de la machine parallèle le travail généré par le programme en exécution. Nous avons deux approches légèrement différentes pour la régulation de charge : l'*équilibrage de charge* qui prévoit une distribution équitable du travail entre les nœuds, et la *répartition de charge* qui cherche à éviter que des ressources soient inactives pendant l'exécution du programme.

2.3.4 Grandeurs caractéristiques des applications

Jusqu'ici nous avons traité l'ordonnancement comme un mécanisme de support à l'exécution. Cependant il est souvent associé à l'obtention d'un gain de performance lors de l'exécution d'un programme parallèle. Par exemple, réduire le temps total d'exécution d'un programme ou ses besoins en mémoire. Cela nécessite de quantifier quelques informations concernant l'exécution des programmes.

Nous listons ci-dessous les grandeurs les plus couramment utilisées pour mesurer les caractéristiques de l'exécution d'un programme parallèle. Un point important est que nous supposons que ses grandeurs sont intrinsèques à l'exécution d'un programme $\mathcal{P}$ pour une entrée x. Nous employons ces grandeurs dans la suite de ce document.

- T_s : durée de l'exécution séquentielle de $\mathcal{P}(x)$.
- T_1 : durée de l'exécution parallèle de $\mathcal{P}(x)$ lors de son exécution sur une architecture à un seul processeur. Nous associons T_1 au travail total réalisée par l'application.
- T_p : durée de l'exécution de $\mathcal{P}(x)$ sur une architecture à p processeurs.
- T_∞ : durée d'exécution de $\mathcal{P}(x)$ sur une infinité de processeurs. Cette grandeur est aussi appelée profondeur ou chemin critique du programme parallèle.

Elle représente le maximum des durées d'exécution des ensembles d'instructions qui doivent être exécutées séquentiellement selon l'ordre défini par les contraintes de précédence (c'est-à-dire du chemin critique).

- S_1 : consommation mémoire de l'exécution de $\mathcal{P}(x)$ sur une architecture à un seul processeur.
- S_p : consommation mémoire de l'exécution de $\mathcal{P}(x)$ sur une architecture à p processeurs.
- C_1 : volume de communications engendrées par l'exécution de $\mathcal{P}(x)$ sur un processeur qui travaille sur un module de mémoire distribué. C_1 correspond alors à une borne supperieure aux accès distants à la mémoire.
- C_∞ : délai ou chemin critique en communications. C'est le plus grand volume de communications réalisé par des tâches selon les contraintes de précédence.

2.4 Vers une interface générique

Cette section introduit le problème de la construction d'un environnement d'exécution parallèle. Dans ce contexte, l'ordonnancement est la brique centrale qui permet d'assurer l'exécution d'une application sur un architecture parallèle.

Dans les sections précédentes nous avons présenté les problèmes associés aux algorithmes d'ordonnancement. Les mécanismes employés pour les résoudre sont bien maîtrisés, de sorte que plusieurs politiques d'ordonnancement ont été proposées[5]. Nous avons aussi identifié les sous-sytèmes internes à un ordonnanceur (décision et manipulation d'information). Une analyse du problème de l'ordonnancement a permis à Willebeek-LeMair et Reeves [127] (voir aussi [125, 126]) et à Jacqmot [73] de caractériser les services offerts par les politiques d'ordonnancement d'une façon générale[6]. Cependant les problèmes liés à l'interaction d'un noyau d'ordonnancement avec le niveau applicatif et le niveau d'architecture ne sont abordés que lors de l'implantation d'un environnement d'exécution parallèle. Or la vision que nous voulons présenter est qu'il est possible de définir les échanges de services entre les niveaux avant de rentrer dans l'implantation de l'environnement.

Comme nous l'avons montré dans ce chapitre, un environnement d'exécution parallèle est composé de trois agents indépendants : l'application, l'architecture et un noyau d'ordonnancement. Le problème qui se pose est de maintenir le niveau d'ordonnancement indépendant lors de l'implantation d'un environnement d'exécution parallèle. C'est-à-dire qu'il est désirable que son code ne soit pas entremêlé

[5] Nous citons [33, 106, 45] pour une vision générale de diverses politiques d'ordonnancement.

[6] Dans le deux cas, les études ont été orientés vers des algorithmes d'ordonnancement dynamiques.

au code de l'application et qu'il ne soit pas très fortement associé aux caractéristiques d'une architecture. L'intérêt est double :

– D'une part, le code applicatif sera plus portable car il est possible d'adapter son exécution à une architecture cible (par exemple 1, 2 ou 1000 processeurs) en changeant la stratégie d'ordonnancement.
– D'autre part, il est possible de réutiliser un même algorithme d'ordonnancement pour différentes applications.

En solution à ce problème nous cherchons à identifier les services requis et offerts par chaque module et à définir des interfaces d'accès à ces services.

Dans la figure 2.2 est représenté l'environnement d'exécution parallèle tel que proposé par Casavant et Kuhl (que nous avons reproduit dans la figure 2.1, page 22). Cependant ici les interfaces de services entre les composants sont mises en évidence afin de caractériser la modularité de l'environnement : les échanges de services sont réalisés par l'interface sans qu'un module n'ait besoin de connaître comment les autres ont été implantés.

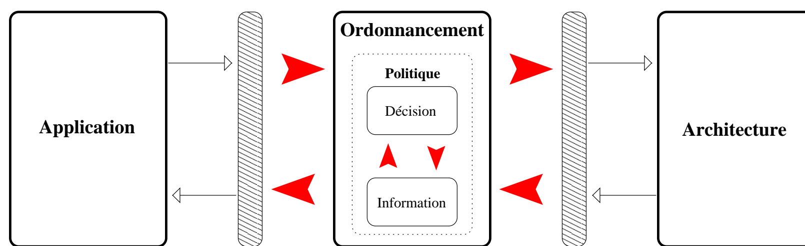

Figure 2.2 *Identification des interfaces entre les modules d'un environnement d'exécution parallèle. Une interface de services permet l'interaction du noyau d'ordonnancement avec les niveaux d'application et d'architecture : le noyau d'ordonnancement est ainsi implanté indépendamment des programmes parallèles et peut être plus facilement porté entre différentes machines.*

De plus, l'utilisation d'interfaces permet d'encapsuler[7] les modules ; ainsi diverses implantations sont possibles pour chacun de ces modules. En effet, l'utilisation d'un module est réalisée par l'accès aux services proposées par son interface, donc de façon totalement indépendante de l'implantation du traitement de ce service [79]. En particulier nous considérons la possibilité d'implantation de diverses

[7]Nous utilisons ici le terme *encapsuler* avec la sémantique des langages orienté objets qui permet de dissocier la mise en œuvre d'un module de son interface [17].

politiques d'ordonnancement en encapsulant le noyau d'ordonnancement par une interface de services.

Nous voulons dans cette thèse étudier chaque composante d'un environnement d'exécution parallèle pour pouvoir definir les interfaces de services nécessaires entre eux. Nous cherchons donc à caractériser la façon dont ces différents modules interagissent [17].

2.5 Conclusion

Nous avons vu dans ce chapitre que l'ordonnancement est un problème lié aux environnements d'exécution parallèle. Son rôle est de supporter l'exécution d'un programme parallèle sur une architecture elle aussi parallèle. Souvent cet ordonnancement est appliqué dans le but d'optimiser un paramètre de performance lors de l'exécution de l'application.

Dans la section 2.4 nous avons envisagé un mécanisme d'interface pour les modules d'un environnement d'exécution parallèle. Avec ce mécanisme nous voulons traiter le problème de l'ordonnancement indépendamment du niveau applicatif et du niveau d'architecture.

Dans le prochain chapitre nous présentons différents environnements d'exécution parallèle contenant un noyau d'ordonnancement proposés dans la littérature. Nous voulons identifier dans ces environnements les relations existant entre un noyau d'ordonnancement et les niveaux applicatif et architecture.

Nous dédions les trois chapitres de la deuxième partie à la discussion de chacune des composantes de l'environnement d'exécution parallèle identifié dans la figure 2.2. Le but est de spécifier les services requis dans l'interface d'un noyau d'ordonnancement.

3

Environnements pour l’ordonnancement

Dans ce chapitre, sont présentés quelques environnements d’exécution parallèle supportant un noyau d’ordonnancement. Pour ces environnements nous cherchons à identifier le mécanisme d’ordonnancement employé et son interface avec les niveaux application et exécution.

Ce chapitre est divisé en quatre sections. Les deux premières, section 3.1 et 3.2, sont dédiées aux environnements d’ordonnancement dynamique. Nous avons choisi de considérer les deux cas d’ordonnancement dynamique, l’équilibrage et le partage de charge, en deux sections différentes bien que, dans la pratique ces deux notions sont assez proches. La section 3.3 présente des environnements supportant des algorithmes statiques. Pour conclure, section 3.4 nous présentons deux tableaux pour résumer les caractéristiques des environnements étudiés.

3.1 Équilibrage de charge dynamique

Dans cette section sont présentés deux environnements pour l’équilibrage de charge sur machines parallèles : GTLB et DPC++. GTLB est un noyau d’ordonnancement pour un modèle de programmation de granularité assez fine basé sur une architecture à mémoire partagée ; pour l’équilibrage de charge, GTLB prend en considération seulement la charge gérée par le programme en cours d’exécution. DPC++, au contraire, est orienté vers des applications de grosse granularité

s'exécutant sur une architecture à mémoire distribuée ; la machine parallèle est supposée non dédiée à l'application, l'information de charge de la machine considère l'influence des autres applications en exécution simultanée.

3.1.1 GTLB

GTLB [38] – « Équilibreur » générique de processus légers[1] – est un noyau d'ordonnancement de processus légers, implanté comme une bibliothèque de fonctions C ; la dernière version distribuée de GTLB date de 1998. Ce noyau est utilisé sur PM² [39] pour supporter l'exécution d'applications du type séparation & évaluation [38].

GTLB comme PM² font partie du projet ESPACE, développé au LIFL de l'Université des Sciences et Technologies de Lille[2].

Expression du parallélisme. Les applications cibles de GTLB sont composées d'un grand nombre d'activités concurrentes qui accèdent des données sur une mémoire commune sans cohérence. Des tâches sont créées dynamiquement et s'exécutent de manière asynchrone – toutes les interations sont réalisées par l'accès à des données dans la mémoire supposée globale[3]. L'ordre de traitement de ces accès suit la politique du premier arrivé premier servi.

L'ordonnancement. Toute tâche créée est immediatement prête à être exécutée et est démarrée. Une politique de placement (choix du processeur) pour le démarrage de la tâche prend en consideration un tableau contenant les informations de charge de tous les nœuds de la machine. Les tâches n'ayant pas la même durée, il peut se présenter qu'un nœud devienne sous-chargé. Un mécanisme de migration de tâches est alors activé de façon à faire migrer d'un nœud surchargé un ensemble de tâches vers un nœud sous-chargé de façon à égaliser la charge entre eux. Un démon est responsable dans un nœud du contrôle de la charge d'un à fin de détecter une situation de sous-charge ; ce démon est une activité interne à l'ordonnancement qui s'exécute concourramment à l'application.

[1] De l'anglais : *Generic Threads Load Balancer*.

[2] Le site WEB du projet ESPACE est : `http://www.lifl.fr/~mehaut/espace.htm`. PM^2 est actuellement en cours de développement au sein du projet ReMaP `http://www.ens-lyon.fr/LIP/ReMaP`, au LIP – Laboratoire d'Informatique du Parallélisme –, Lyon.

[3] Seule une version restreinte de cette mémoire globale adaptée aux problèmes de séparation & évaluation a été implantée en pratique

Les informations de charge. La charge d'un nœud reflète le nombre de tâches qui sont en cours d'exécution sur le site. Toutes les tâches sont considérées identiques et leurs durées ne sont pas prises en compte. Le tableau représentatif de la charge globale est maintenu de façon distribuée : chaque nœud maintient sa propre information de charge. Lorsque le nombre de tâches en exécution varie de δ, le nœud envoie un message contenant l'information de sa charge aux autres nœuds[4]. Pour cet envoi, il est considéré que les nœuds sont organisés en une structure d'anneau. Le message est envoyé à un voisin, disons le voisin de « gauche », de sorte que ce voisin puisse comparer sa charge avec celle reçue. Selon le résultat de cette comparaison, soit une procédure de migration est déclenchée, soit le message est renvoyé sur l'anneau – si le message arrive sur son nœud d'origine, il est détruit.

3.1.2 DPC++

DPC++ [26, 30] – Programmation Distribuée en C++[5] – est un support pour la programmation parallèle sur des réseaux de stations de travail. Le modèle de programmation prévoit une extension du langage C++ permettant la création des objets distribués instanciés sur des processus autonomes dédiés. Une première version. Le projet DPC++ a débuté en 1991 dans le CPGCC de l'Université Federale du Rio Grande do Sul ; une première version a été distribuée en 1994. Actuellement le concept de multiples fils d'exécution est en train d'être introduit dans le modèle pour réduire sa granularité [4, 3] et parallèlement les problèmes de la tolérance aux pannes sont étudies[6].

Expression du parallélisme. En DPC++ le parallélisme est exprimé sous la forme d'objets communicants : chaque objet contient un ensemble de données et un interface d'accès aux services exécutés sur ces données. Les échanges entre les objets se font par des messages asynchrones ou synchrones : un message consiste en une invocation d'une méthode sur l'objet récepteur – une méthode sans retour de données (c'est-à-dire, `void`) implique donc un message asynchrone. Aucun contrôle de la cohérence au niveau de l'ordre d'exécution n'est réalisé, les invocations sont traitées en exclusion mutuelle selon l'ordre d'arrivée sur l'objet.

L'ordonnancement. L'ordonnancement dans DPC++ est réalisé au niveau de l'objet et pas au niveau de l'appel de méthode. Toutes les demandes de création

[4]Dans l'implantation de GTLB la valeur δ est configurable.

[5]De l'anglais : *Distributed Programming in C++*.

[6]Le site WEB du projet DPC++ est : `http://www.inf.ufrgs.br/gpesquisa/procpar/dpc/dpc.html`.

d'objets distribués sont envoyées à un module central, nommé *Répertoire*, responsable du placement de l'objet de façon à ne pas surcharger un nœud du réseau. La fonction de choix du site pour abriter un nouvel objet distribué créé est préalablement fournie au Répertoire ; cette fonction de placement utilise un tableau décrivant la situation en relation à la charge de chaque nœud. Une fois que le placement d'un objet distribué a été réalisé, toutes les invocations de méthodes sont envoyées directement par l'appelant. Dans le cas d'une station de travail devenant saturée, la situation est détectée par le Répertoire qui déclenche un mécanisme de migration d'objets. Ce même mécanisme a été employé aussi pour supporter la tolérance aux pannes [98]. L'ordonnancement des processus sur les processeurs du nœud est laissé à la responsabilité du système d'exploitation natif.

Les informations de charge. Les information de charge de chaque nœud de la machine sont centralisées dans un tableau maintenu sur le Répertoire. L'opération d'actualisation de ce tableau profite des informations de charge obtenues par des objets *espions* qui s'exécutent sur chaque nœud dans le but unique de scruter l'utilisation des ressources, notament mémoire et pourcentage d'utilisation du processeur. L'envoi de l'information de charge d'un nœud est déclenché par une variation dans l'indice d'utilisation enregistré par l'espion.

3.2 Partage de charge dynamique

Dans cette section sont présentés cinq environnements de partage de charge dynamique. Dans ces environnements, nous en présentons trois qui permettent un certain contrôle d'exécution par l'abstraction d'une sémantique d'exécution : Cid, Cilk et Jade. Tant dans Cilk que dans Jade l'ordonnancement est totalement transparent au programme, alors que Cid force le programmeur à interagir avec le noyau d'ordonnancement, notamment au moment de la création de nouvelles activités.

Millipede propose une autre méthode de régulation de charge qui ne prend pas en compte les contraintes de précédence entre les activités, mais qui au contraire des Cid, Cilk et Jade, considère la charge globale de la machine et pas seulement la charge générée par une application.

Finalement, un cinquième environnement nommé Dynamo permet au programmeur de définir sa propre politique de régulation de charge basée sur un algorithme de liste.

3.2.1 Cid

Cid [93] est une extension de C, développée par le laboratoire de recherche de Digital à Cambridge, pour la programmation parallèle sur des machines à mémoire distribuée orientée vers des applications manipulant des structures de données recursives, du type arbres, listes et graphes. Sa première version date de juin 1995. Un programme Cid en exécution consiste en un ensemble de tâches en exécution concurrente accédant une mémoire commune et échangant des données par l'envoi de paramètres aux tâches et par retour de résultats[7].

Expression du parallélisme. Dans Cid la principale façon de déclarer le parallélisme est l'utilisation de mécanismes du type *fork* / *join*. Après un appel à une opération *fork*, les tâches créatrice et créée sont exécutées concurremment, l'opération *join* est optionnelle et permet à la tâche créatrice d'attendre la terminaison d'une tâche créée. Il est aussi possible de faire un *join* de plusieurs tâches créées. Entre les tâches existe la notion de mémoire commune ; cette mémoire est maintenue distribuée parmi les modules de mémoire privés à chaque processeur. Un niveau de cohérence est supporté dans cette mémoire globale par l'utilisation d'une structure de données propre à Cid, permettant des accès protégés par des primitives d'exclusion mutuelle (verrou).

L'ordonnancement. Un premier niveau d'ordonnancement emploie des multiples flots d'exécution concurrents sur un processeur pour masquer les temps d'attente résultants des accès à la mémoire commune. Sur ce noyau, l'ordonnancement global est activé en réaction à une opération réalisée par le programme en exécution, notamment la création d'une tâche ou l'exécution d'un *join* ou lorsqu'un processeur devient inactif. Cid laisse le programmeur indiquer quel type de traitement doit être employé pour déterminer le placement des tâches créées ; quatre différents types de primitives *fork* sont offertes :

1 *fork* avec contrôle de charge : le système trouve le processeur le moins chargé pour exécuter la nouvelle tâche.

2 *fork* avec contrôle de migration de charge : lorsqu'aucun processeur n'est inactif, la tâche n'est pas créée et l'exécution du service est réalisée comme un simple appel de fonction ; dans le cas où un processeur est inactif, la tâche est créée et ensuite migrée à ce processeur.

3 *fork* avec contrôle de localité : c'est un *fork* où est indiquée la donnée dans la mémoire globale qui sera manipulée par la nouvelle tâche. Si la donnée est sur la

[7] Le site WEB de Cid est maintenu dans le site de Digital : `http://www.crl.research.digital.com`.

mémoire privée du processeur, la tâche est exécutée dans le même flot d'exécution que la tâche créatrice ; dans le cas contraire un nouveaux flot d'exécution est créé sur le processeur qui possède la donnée.

4 *fork* avec contrôle de granularité : le système est chargé de définir le nombre de tâches qui doivent être créées pour exécuter le traitement d'un ensemble de données de taille N, de façon à avoir un nombre M de tâches, tel que $1 \leq M \leq N$.

Lors d'une opération de *join* la tâche est bloquée dans l'attente d'une synchronisation avec d'autres tâches. Dès que cette synchronisation est satisfaite, la tâche est insérée dans une liste de réserve, en attendant son redémarrage.

Dans le cas où un processeur devient inactif, un mécanisme de vol de travail permet d'obtenir une nouvelle tâche. Le principe de cet algorithme dans Cid est de choisir un processeur victime et de lui demander une tâche. Ce processeur victime, au recevoir la requête de travail, envoie en réponse une tâche de sa liste de réserve, où, à défaut lors de l'exécution d'un *fork* du type 2 (*fork* avec contrôle de migration de charge) ; en effet il n'existe pas de mécanisme permettant la préemption et la migration de tâches en cours d'exécution.

Les informations de charge. L'unique information de charge est le nombre de tâches en cours d'exécution sur les processeurs. Cette information est maintenue individuellement sur chaque processeur et utilisée pour détecter une situation d'inactivité.

3.2.2 Cilk

Cilk [14] est un langage parallèle développé à partir de 1993 au MIT ; actuellement il est disponible dans sa cinquième version[8]. Cilk est basé sur une extension du langage C pour supporter un parallélisme fonctionnel du type *fork* / *join*. La création des tâches est explicite et, pendant son exécution, elles accèdent une espace mémoire commun. Ces fonctions concurrentes sont ainsi traduites en tâches qui, à l'exécution, accèdent une zone de mémoire commune. Cette situation est analogue à un classique retour de procédure ; Cilk optimise la pile pour que la création et le retour en fin de tâche ne soient guère plus coûteux qu'un classique appel puis retour de fonction.

Expression du parallélisme. Le parallélisme en Cilk est spécifié à l'aide du mot clé `spawn`. Un `spawn` placé devant un appel de fonction implique, lors de

[8]Le site WEB de Cilk est `http://supertech.lcs.mit.edu/cilk`.

l'exécution, la création d'une tâche pour évaluer cette fonction. La sémantique de cet appel diffère de celle d'un appel classique de fonction au sens où la procédure appelante peut continuer son exécution en parallèle de l'évaluation de la fonction appelée – au lieu d'attendre son retour pour continuer. Cette exécution étant asynchrone, la procédure créatrice ne peut pas utiliser le résultat de la fonction appelée sans une synchronisation explicite, c'est qui est possible par utilisation de l'instruction `sync`. Cette instruction a pour effet d'attendre la terminaison de toutes les fonctions appelées en parallèle par la fonction mère avant ce `sync` : le parallélisme exprimé est donc de type série-parallèle. Les tâches sœurs créées sont supposées indépendantes : il y a donc risque de concurrence sur les accès à la mémoire partagée, concurrence qui doit être gérée par l'utilisateur.

En plus de cette génération de parallélisme, le langage Cilk permet d'associer lors de la création d'une tâche une fonction « réflexe » ou *callback* qui sera exécutée en exclusion mutuelle lors de la terminaison de cette tâche. Il est possible à l'intérieur de cette fonction de demander la destruction de l'ensemble des tâches créées par la procédure mère et non encore exécutées, possibilité communément utilisée dans les algorithmes parallèles de recherche spéculative.

L'ordonnancement. Lors de l'exécution, Cilk suppose l'existence d'un espace mémoire unique, globalement accessible sans synchronisation par tous les processeurs. D'ailleurs Cilk a été initialement développé sur des architectures du type SMP [74] ; puis la version pour architectures à mémoire distribuée, proposée dans [100], est basée sur l'implantation d'un mécanisme de pagination pour émuler un espace mémoire unique. La base de l'algorithme d'ordonnancement est le vol de travail. Sur chaque processeur de la machine, un flot d'exécution est supporté. Sur ce flot, les tâches sont exécutées selon une politique profondeur d'abord : à chaque création d'une tâche, l'ordonnancement est activé pour bloquer la tâche créatrice et démarrer la tâche créée. En effet la tâche créatrice est considérée finie et une nouvelle tâche pour la suite est créée ; d'une façon semblable, une tâche est terminée lors de l'exécution d'une instruction de `sync` : une tâche non prête est créée pour la suite de l'exécution après que la condition de synchronisation soit satisfaite. Les tâches prêtes sont maintenues dans une pile, de sorte que à la fin d'une tâche, la tâche au sommet de cette pile est redémarrée. Lorsque cette pile est vide le processeur devient inactif. Dans une telle situation, une tâche est « volée » de la pile de tâches bloquées d'un autre processeur choisi au hasard. La tâche choisie pour être volée est celle présente dans la position la plus basse de la pile, c'est-à-dire celle qui, sur le processeur volé, serait la dernière à être exécutée. Le choix de cette tâche est justifié par le fait que cette tâche est probablement celle qui a la plus grande longueur.

Dans la version distribuée [100], orientée vers un réseau de nœuds de SMP, le

principe du vol de travail a été maintenu, ainsi que le déterminisme de l'exécution. Pour maintenir le déterminisme, il a été nécessaire de garantir l'intégrité de la mémoire commune aux tâches. Un mécanisme de cache a été introduit, ce qui a impliqué une augmentation du surcoût de l'ordonnancement. Pour réduire l'impact de ce surcoût, l'ordonnancement prend en consideration une notion de « localité » : pour effectuer le vol le choix du processeur victime n'est pas totalement aléatoire, les processeurs sur le même nœud SMP, qui partagent la même mémoire physique, ont une plus grande probabilité d'être choisis.

Les informations de charge. Dans la politique d'ordonnancement de Cilk, il n'existe pas explicitement d'information de charge du processeur. Cet information est en réalité obtenue implicitement par l'activité ou inactivité du processeur : soit le processeur a du travail, soit il en n'a pas.

3.2.3 Millipede

Le projet Millipede [72, 107], en développement par l'Israel Institute of Technology, a comme but le développement de technologies de hautes performances pour permettre l'exécution distribuée d'applications. Un langage n'est pas proprement defini par Millipede, mais un environnement de compilation et d'exécution utilisable, par l'intermède des interfaces applicatives, par des langages tels que HPF et Java. Millipede est attaché à un programme d'application sous la forme d'une bibliothèque. Le modèle de programmation est basé sur des tâches communicantes et une mémoire partagée.

Millipede construit une machine parallèle virtuelle construite sur un cluster de stations non dédiées sur lequel est implanté un mécanisme de gestion d'une mémoire globale[9].

Expression du parallélisme. Des tâches sont créées et exécutées de façon concurrente en utilisant une mémoire globale pour l'échange de données. La création de tâches implique son démarrage immédiat, qui peut, selon la spécification fournie à la construction de la tâche, être soit sur le nœud qui a exécuté l'opération de création soit sur un nœud distant. Des messages peuvent être envoyés entre les tâches pour permettre la synchronisation de leurs exécutions. La mémoire globale consiste en un ensemble de pages maintenues d'une façon distribuée sur les modules de mémoire des nœuds de la machine parallèle. Le mécanisme d'ordonnancement ne garantit aucune forme de cohérence, à part celle introduite explicitement

[9] Le site WEB du projet Millipede est : `http://dsl.cs.technion.ac.il/Millipede`.

par l'utilisation de synchronisation. Un autre niveau de cohérence est maintenu au niveau de la gestion de la mémoire distribuée, mais il n'est pas considéré ici.

L'ordonnancement. L'ordonnancement dans Millipede est basé sur le principe de la migration de tâches. Le but de l'ordonnancement consiste à réguler le nombre d'activités en exécution sur chaque nœud de la machine, tout en prenant en compte les coûts de communication engendrés par les accès aux données dans la mémoire globale. C'est le mécanisme de gestion de la mémoire globale qui permet de comptabiliser les accès réalisés pour chaque tâche à chaque donnée ; en utilisant cette information, l'ordonnancement a le moyen de choisir les plus aptes à migrer d'un nœud surchargé vers un nœud sous-chargé. Une caractéristique de l'implantation de l'ordonnancement est la complète indépendance de fonctionnalités entre la politique de déclenchement des opérations de régulation de charge et la politique de choix des tâches qui seront migrées. Cette indépendance peut être verifiée par la possibilité qui est donnée au programmeur de définir sa propre politique de déclenchement du processus de migration, en determinant, sur un nœud, la quantité des tâches à migrer et le nœud qui doit les recevoir. Toutefois, la politique de choix des tâches à migrer est toujours réalisée par l'environnement, car lui seulement peut interagir avec le module de gestion de la mémoire globale ; il peut ainsi choisir les tâches en fonction de leur accès à la mémoire global, en exploitant la localité des données. Pour le choix des tâches à migrer, le surcoût généré par les échanges de messages des tâches communicantes est aussi considéré.

Il est important de noter que, l'implantation de Millipede consiste en un ensemble de démons qui s'exécutent sur chaque nœud de la machine parallèle. Le réseau qui supporte l'exécution n'étant pas dédié à une application, plusieurs programmes Millipede peuvent être en cours d'exécution en même temps. Toutefois, les démons ne sont pas repliqués pour chaque programme : un démon lancé sur un nœud supporte la régulation des tâches générées pour tous les programmes. Millipede exploite aussi le temps pendant lequel un nœud de ce réseau n'est pas utilisé de façon interactive par son « propriétaire ».

Les informations de charge. À la base, les informations de charge manipulées sont le nombre de tâches en exécution sur le nœud et les informations des accès aux pages de la mémoire globale. Tant que les informations concernant les accès à la mémoire globale sont utilisées seulement au niveau du nœud pour le choix des tâches à migrer elles ne sont pas repliquées sur tous les nœuds de la machine. Les informations concernant le nombre de tâches en exécution sur chaque nœud sont, au contraire, maintenues sur chaque nœud, de façon à permettre à chaque nœud d'activer une opération de migration.

Une information de charge additionnelle gérée au niveau d'ordonnancement permet de détecter si un nœud de la machine parallèle sert à un utilisateur de façon interactive. Dans ce cas, ce nœud est consideré comme non apte à recevoir des tâches.

3.2.4 Jade

Jade [102, 101] est une extension de C pour la programmation parallèle developpée à l'Université de Standford ; la dernière version de Jade date de 1994[10]. Jade étend le langage séquentiel C avec une notion de bloc d'instructions. Chaque bloc est annoté par les données accédées dans le bloc et le droit d'accès (lecture ou écriture). Le parallélisme est implicite. Lors de l'exécution, l'entrée dans un bloc est interprétée comme la création d'une tâche dont les entrées et sorties sont identifiées. Ainsi il est possible de modéliser l'exécution d'un programme Jade par un graphe de flot de données. La sémantique de Jade est que toute lecture voit la dernière écriture relativement à l'ordre séquentiel d'exécution. Pour implanter cette sémantique, le noyau exécutif de Jade gère un graphe de précédence distribué.

Expression du parallélisme. Un programme Jade consiste en un programme C dans lequel sont rajoutées des directives d'exploitation du parallélisme. Il permet, à l'image de Cilk, de créer des tâches associées à des flots d'exécution indépendants. Dans chaque tâche sont utilisés des opérateurs qui identifient les accès que la tâche réalise sur les données de la mémoire globale ; sont notamment identifiées les données lues et écrites par la tâche. Cependant il faut remarquer que, au contraire de Cilk, une tâche créée n'est pas forcément une tâche prête : elle pourra être non-prête si elle est déjà créée mais que une (ou plusieurs) tâches qui la précèdent dans le graphe ne sont pas finies, c'est-à-dire, la (ou les) données dont elle a besoin en entrée ne sont pas disponibles. Ainsi la concurrence d'exécution entre les tâches est limitée par les accès aux données. Dans le cours de son exécution, une tâche peut changer les modes d'accès sur les données de la mémoire globale, mais ce cas n'est pas consideré ici.

L'ordonnancement. L'ordonnancement de Jade est centralisé et profite des informations contenues dans le graphe de flot de données pour exploiter la localité des données. Il est basé sur une liste de tâches triée selon les références faites à des données dans la mémoire globale. Une tâche créée est insérée dans cette liste en

[10] Des informations complementaires sur Jade sont disponibles sur la WEB dans les adresses : `http://suif.stanford.edu/index.html` et `http://www.cag.lcs.mit.edu/~rinard/jade`.

fonction des données qu'elle manipule ; le placement des tâches se fait implicitement dans cette opération. Un processeur qui finit l'exécution d'une tâche cherche parmi les tâches prêtes celle qui accède la donnée manipulée par la tâche qui vient de terminer. Une fois démarrée une tâche s'exécute jusqu'à sa terminaison[11]. Au cas où aucune tâche n'est trouvée, un autre objet ayant au moins une tâche prête est récuperé par le processeur pour son exécution – il s'agit, en quelque sorte, d'une opération de migration de tâches. Deux versions de Jade ont été construites, une pour une machine du type SMP et une autre pour des architectures à mémoire distribuée. Dans le deuxième cas nous avons une gestion distribuée de la liste de tâches ; des messages sont pourtant échangés entre les nœuds pour actualiser l'état des tâches associées à chaque donnée de la mémoire globale.

Les informations de charge. Les informations de charge sont essentiellement le nombre de tâches prêtes sur un processeur et la localité des données. Le nombre de tâches prêtes est considéré implicitement : soit le processeur est actif, c'est-à-dire il exécute une tâche, soit il ne l'est pas. Cette information est maintenue localement par le processeur et l'absence de tâches prêtes implique le déclenchement des opérations permettant la migration de tâches. L'information de localité des données est aussi maintenue sur chaque processeur : (*i*) lors du choix d'une nouvelle tâche à exécuter sur un processeur la préférence est donnée à une tâche qui accède la même donnée que la tâche qui vient de terminer ; (*ii*) au moment de choisir les tâches à migrer, le choix est réalisé parmi toutes les tâches associées à une donnée de la mémoire globale.

3.2.5 Dynamo

Dynamo [119] consiste en une bibliothèque de fonctions qui permet de séparer l'ordonnancement d'une application de sa programmation. Ce projet a été mené dans l'Université de Umea. La bibliothèque Dynamo est divisée en deux groupes de fonctions : celles destinées à la programmation de l'application et celles destinées à la gestion de la régulation de charge. L'interface entre l'application et l'ordonnancement permet de changer facilement les politiques d'ordonnancement : changer une politique signifie changer la sémantique associée à une des fonctions de la bibliothèque de régulation de charge. Ces changements sont réalisés sans avoir à modifer le code de l'application. Un ensemble de politiques de régulation de charge est fourni avec la bibliothèque.

[11] Rappelons que la situation où la tâche modifie les droits d'accès aux données n'est pas considérée.

Expression du parallélisme. Le parallélisme dans Dynamo est décrit par la création de tâches. À chaque tâche peut être associé un ensemble de données sur lesquelles la tâche a libre accès, tant pour la lecture que pour l'écriture. Un même ensemble de données peut être partagé par plusieurs tâches ; dans ce cas, il n'est garanti par Dynamo aucun contrôle de cohérence des accès à cette zone de mémoire. Dynamo offre toutefois la distinction entre le moment de la création d'une tâche et son démarrage ; le programmeur peut ainsi implanter sa propre méthode de cohérence.

L'ordonnancement. L'exécution des programmes est réalisée par un ensemble de p processeurs dotés d'une espace de mémoire locale ; chaque processeur exécute une à une les tâches utilisateurs. Le principe de base de l'ordonnancement est de laisser au programmeur de l'application la responsabilité de définir les actions à réaliser en réponse à chaque événement. Par exemple, au moment où un processeur termine l'exécution d'une tâche, il doit être capable de déterminer qu'une nouvelle tâche peut être démarrée ; si localement, sur le processeur, aucune tâche ne peut être lancée, une recherche de service peut être lancée sur les autres processeurs. Dynamo offre l'ensemble de primitives nécessaires à l'interaction entre le programme en application et le noyau d'ordonnancement. Notamment il est possible de créer des tâches, de gérer ces tâches dans des structures de type liste, de contrôler les tâches créées par l'application, d'envoyer des requêtes de tâches entre les processeurs et de migrer (restreinte) des tâches.

Dynamo permet aussi d'associer à chaque tâche un ensemble d'attributs. Ces attributs décrivent les caractéristiques des tâches, telles que son *coût* et sa *priorité* d'exécution par rapport aux autres tâches, la *taille* de son ensemble de données et sa *localisation*, c'est-à-dire le module dans lequel ses données sont stockées. Dans l'implantation d'une politique de régulation de charge ces attributs peuvent êtres pris en compte.

Les informations de charge. À l'image de la politique de régulation , le contrôle des informations de charge est de la responsabilité de la politique implantée. Dynamo offre des primitives pour comptabiliser des indices et de les échanger entre deux processeurs.

3.3 Ordonnancement statique

Dans cette section nous présentons deux techniques de régulation de charge statique. Dans ce cas, il est considéré que le graphe de tâches est construit et soumis

à l'outil d'ordonnancement avant le démarrage de l'exécution. Les outils que nous considérons ici interprètent différemment le graphe : tandis que PYRROS et RAPID interprètent les arêtes comme des précédences entre les tâches, Metis et SCOTCH interprètent les arêtes comme des partages de données entre les tâches.

3.3.1 PYRROS/RAPID

PYRROS [130] est une bibliothèque comportant un langage de description d'un graphe de tâches ainsi que des fonctions pour son ordonnancement. Dans cet environnement un programme est représenté par un ensemble de tâches soumises à des relations de dépendance. RAPID [56] est un environnement reposant sur les mêmes principes que PYRROS mais qui se différencie par l'expression du parallélisme, basé sur l'accès aux objets partagés, et par le support d'exécution des applications[12].

Expression du parallélisme. L'expression du parallélisme est à la charge de l'utilisateur. Dans PYRROS un langage de description du graphe de précédence est fourni permettant de décrire les tâches et les transferts de données du programme, ainsi que leur coût respectif. Toutes les données accédées par une tâche doivent être disponibles à son démarrage ; la tâche s'exécute ensuite sans autre synchronisation.

Dans RAPID le graphe de précédence est construit de façon implicite par l'analyse des accès aux objets partagés effectués par les tâches. La description des objets partagés et le partitionnement en tâches restent à la charge de l'utilisateur.

L'ordonnancement. L'ordonnancement du graphe de précédence est basé sur l'utilisation de l'algorithme DSC de regroupement[13]. Les groupes de tâches obtenus sont ensuite alloués aux processeurs de la machine cible par des techniques d'équilibrage de charge ; un algorithme dû à Bokhari permet de réduire le volume des communications. Une fois l'allocation des tâches fixée, l'ordre d'exécution sur chaque processeur est décidé par un algorithme de liste basé sur les premières tâches prêtes. Les tâches sont démarrées sur chaque processeur, une à une, selon l'ordre specifié par cette liste.

Il est à noter que RAPID propose également une politique d'allocation *owner-computes rule*, qui place une tâche sur le même site que l'un de ses objets accédés en écriture. Dans un article ultérieur [128], T. Yang et C. Fu ont présenté d'autres politiques d'ordonnancement prenant en compte la consommation mémoire de l'application.

[12]D'autres informations sur PYRROS et RAPID sont disponibles dans la page WEB de son auteur : `http://www.cs.ucsb.edu/~tyang`.

[13]L'algorithme DSC est présenté dans la section 4.2.1.3, page 64.

Les informations de charge. Les informations manipulées par PYRROS sont celles annotées dans le graphe de précédence : coût des tâches et volume de données échangées. PYRROS utilise également une description de l'architecture cible, spécifiant la cadence des processeurs – la machine est supposée homogène –, la vitesse du réseau de communication et la topologie d'interconnexion.

3.3.2 Metis et SCOTCH

Metis[14] [78] et SCOTCH[15] [96] sont des bibliothèques de fonctions permettant de découper en groupes un graphe de tâches non-orienté de façon équilibré, en minimisant les arrêtes reliant des tâches de groupes différents[16]. Un modèle de programme qui exploite ces outils est composé de tâches qui s'exécutent d'une façon asynchrone. Dans le graphe, les nœuds représentent les tâches construites et les arêtes, non orientées, les données échangées entre deux tâches.

Expression du parallélisme. Toutes les tâches sont créées au début de l'exécution et démarrées immédiatement. Au début de l'exécution les données sont aussi connues ainsi que les accès réalisés sur ces données par les tâches. Étant donné que les tâches s'exécutent d'une façon concurrente sans synchronisation, la sémantique d'accès n'est pas gérée.

L'ordonnancement. L'ordonnancement profite de l'information fournie par le graphe : les tâches à exécuter et les données sont partagées de façon à identifier des groupes de tâches qui travaillent sur un même ensemble de données. Le résultat est l'obtention d'une représentation résumée du graphe initial. Une deuxième phase d'ordonnancement distribue les groupes sur les nœuds de la machine parallèle. Dans cette deuxième phase, Metis considère une machine homogène reliée par un réseau complet ; SCOTCH considère une machine hétérogène sur un réseau quelconque – cette machine est elle aussi représentée par un graphe. Il est à noter que l'exécution des tâches n'est gérée ni par Metis ni par SCOTCH ; ils ne fournissent que le placement devant être réalisé ; un support exécutif doit être employé.

Les informations de charge. Les informations prises en compte pour réaliser la partition du graphe sont celles annotées sur le graphe : le coût d'exécution des

[14]La page WEB officielle de Metis est : `http://www-users.cs.umn.edu/~karypis/metis/metis.html`.

[15]La page WEB officielle de SCOTCH est : `http://www.labri.u-bordeaux.fr/Equipe/ALiENor/membre/pelegrin/scotch`.

[16]Une implantation parallèle de Metis, ParMetis [77], est aussi disponible.

tâches et les coûts engendrés par les communications dûes aux accès aux données. De plus, SCOTCH emploie la description de la machine en fonction de ces performances. Par cette description, SCOTCH prend en compte la puissance de calcul de chaque nœud de la machine et la vitesse du réseau de communication.

3.4 Conclusion

Nous retrouvons dans les tableaux 3.1 et 3.2 un récapitulatif des caractéristiques de chacun des environnements que nous avons présentés dans ce chapitre.

Tableau 3.1 *Modèles de programmation des environnements étudiés : un récapitulatif des caractéristiques.*

	Synchronisation	Description du parallélisme	Données
Création dynamique des tâches			
GTLB	Non	Tâches indépendantes	Globales
DPC++	Explicite	Série-parallèle	Locales
Cid	Explicite	Arbre de création de tâches	Globales
Cilk	Explicite	Série-parallèle	Globales
Millipede	Non	Tâches indépendantes	Globales
Jade	Implicite	Graphe de flot de données	Partagées
Création statique des tâches			
PYRROS/RAPID	Explicite	Graphe de précédence	Globales
Metis/SCOTCH	–	Graphe de dépendance	Partagées

Dans le tableau 3.1 nous retrouvons les caractéristiques liées au modèle de programmation des environnements étudiés. Dynamo n'est pas présent sur ce tableau ; en effet, sa particularité est de prévoir un environnement dans lequel le programmeur peut employer les méthodes qu'il souhaite pour implanter son application et pour en réaliser l'ordonnancement. Nous pouvons observer que lorsque le niveau applicatif permet une sémantique d'exécution, un graphe orienté est utilisé pour contrôler les précédences entre les activités. Dans la plupart des cas où les tâches sont générées dynamiquement, nous observons que ce graphe est créé directement à partir des instructions exécutées par le programme (par exemple le `spawn` et le `sync` de Cilk) ; cependant un graphe peut être créé aussi d'une manière implicite, comme dans Jade, par une analyse du flot de données entre les tâches.

Ce graphe peut être en outre exploité par une stratégie d'ordonnancement. C'est le cas des outils d'ordonnancement statiques (PYRROS, RAPID, Metis, SCOTCH) dont l'ordonnancement est calculé à partir de l'exploitation de ce graphe. Nous avons vu cependant qu'un graphe représentant l'exécution d'un programme peut être aussi créé dynamiquement (Cilk, Jade). Dans ce cas, le noyau d'ordonnancement utilise ce graphe pour obtenir quelques informations utiles à une heuristique

d'ordonnancement. Par exemple la localité des données est exploitée dans Jade et la durée des tâches dans Cilk.

Dans le tableau 3.2 nous retrouvons les caractéristiques relatives à la relation entre le niveau d'ordonnancement et l'architecture. Nous observons qu'un noyau exécutif externe, pour la réalisation de l'ordonnancement système, est souvent employé. Nous observons aussi que lorsqu'un environnement fait l'usage de la migration, un mécanisme de préemption est associé, à l'exception de DPC++. Pour DPC++ l'unité manipulée n'est pas vraiment l'activité lancée lors d'un appel d'un méthode, mais un objet entier ; ainsi, la migration correspond au déplacement d'une donnée lorsqu'aucune de ses méthodes n'est en cours d'exécution.

Un dernier point à considérer est l'aspect généraliste proposé par Dynamo. Cet outil a été conçu pour permettre l'implantation de stratégies d'ordonnancement selon les besoins spécifiques des applications. Il est cependant restreint a des stratégies basées sur des algorithmes de liste.

Dans la deuxième partie de ce document nous analysons plus finement les relations d'un noyau d'ordonnancement avec les niveaux applicatif et architecture.

Tableau 3.2 *Récapitulatif des caractéristiques d'ordonnancement des environnements étudiés. DU = Défini par l'Utilisateur.*

	Architecture cible	**Critère de charge de la machine**	**Critère de charge de l'application**	**Préemption**	**Migration**	**Ord. système**
GTLB	Distribuée	DU	DU	Oui	Après démarrage	Marcel
DPC++	Distribuée	Oui	Non	Non	Restreinte	Unix
Cid	SMP et Distribuée	Non	Nombre de tâches ; localisation des données	Points de synchronisation	Restreinte ou tâches bloquées	Unix + processus légers
Cilk	SMP et Distribuée	Non	Nombre et durée des tâches	Points de synchronisation	Restreinte	Unix
Millipede	Distribuée	Oui	Localisation des données	Oui	Après démarrage	Windows/NT
Jade	SMP et Distribuée	Non	Nombre de tâches ; localisation des données	Non	Restreinte	Unix + PVM
Dynamo	Distribuée	DU	DU	Non	Restreinte	–
PYRROS RAPID	Distribuée	Description statique	Coûts annotés			PVM, MPI
Metis	Distribuée	Non	Coûts annotés			Support extérieur
SCOTCH	Distribuée	Description statique	Coûts annotés			Support extérieur

Deuxième partie

Une interface générique pour l'ordonnancement

Avant propos

Nous avons vu dans les chapitres précédents que l'ordonnancement consiste en une couche logicielle qui contrôle l'exécution d'un programme parallèle sur les ressources d'une architecture parallèle. Nous avons vu aussi que l'ordonnancement est indépendant tant du niveau de l'application comme du niveau d'architecture ; il est donc possible de structurer un environnement de régulation en trois couches. Le but de cette partie est alors de spécifier les interfaces entre ces couches.

Dans cette deuxième partie nous allons étudier les échanges de services entre le module d'ordonnancement et les modules d'application et d'architecture. À partir de cet étude, nous aboutissons à la définition d'un noyau d'ordonnancement pour un environnement d'exécution parallèle. Deux caractéristiques sont fondamentales dans ce travail de spécification :

1. *Modularité* : nous voulons un noyau d'ordonnancement qui soit complètement séparé du niveau d'application et du niveau d'architecture. Pour cela nous définissons les interfaces de services nécessaires entre le noyau d'ordonnancement et les deux autres modules.
2. *Généricité* : nous voulons obtenir un noyau d'ordonnancement qui permette d'implanter un ensemble assez large de politiques d'ordonnancement.

La présentation de cette partie est organisée comme suit.

Le chapitre 4 est consacré à l'interface entre l'application et le module de régulation. Nous proposons une interface basée sur l'exploitation à la volée du graphe de flot de données qui permet d'abstraire l'exécution du programme. Une telle interface permet en effet la mise en œuvre de divers ordonnancements, statiques, dynamiques ou hybrides.

Puis le chapitre 5 est consacré à une étude du niveau d'architecture. Nous définissons le comportement espéré d'une machine parallèle et finalement nous présentons l'interface entre le noyau d'ordonnancement et la couche d'exécution.

Finalement, dans le chapitre 6, nous présentons un noyau générique permettant l'implantation de différentes politiques d'ordonnancement.

4

Fondements des stratégies d'ordonnancement applicatif

Ce chapitre est consacré à une présentation des algorithmes utilisés pour l'ordonnancement d'applications. De nombreux algorithmes d'ordonnancement ont été proposés et sont très utilisés de manière pratique. Dans ce chapitre, après une introduction aux différents algorithmes considérés, nous en étudions quelques uns qui sont utilisés de manière pratique : statiques (section 4.2), dynamiques (section 4.3) et enfin mixtes[1] (section 4.4). Nous insistons sur l'intérêt des algorithmes dits de partage de charge qui reposent sur une justification théorique forte (section 4.3.3) lorsque les communications ne sont pas prises en compte.

Le choix d'un algorithme d'ordonnancement pour une application dépend non seulement des caractéristiques de l'application elle-même mais aussi de celles de la machine cible. Même lorsque l'on suppose les processeurs identiques et dédiés à l'application (cadre dans lequel se situe ce chapitre), l'ordonnancement peut être adapté selon la capacité du réseau d'interconnexion ou la capacité mémoire disponible par exemple.

Pour permettre à une même application d'être exécutée sur différentes architectures donc avec un algorithme d'ordonnancement quelconque (qui fait souvent appel à différentes techniques statiques et dynamiques comme nous le verrons au cours de ce chapitre), il est nécessaire de séparer l'algorithme d'ordonnancement du programme applicatif dans deux modules distincts, en définissant leurs interactions.

[1] On dit aussi parfois *hybride*.

En constatant qu'un point commun entre tous les différents algorithmes d'ordonnancement réside dans la description plus ou moins partielle de l'exécution de l'application, nous introduisons dans la section 4.5 le graphe de flots de données *macroscopique* et *dynamique* comme une abstration possible permettant de synthétiser les interactions entre un algorithme d'ordonnancement d'une part et l'application d'autre part. Finalement, dans la section 4.6, nous précisons les interfaces entre ces deux modules.

Le fait de considérer le graphe de flots de données comme une abstraction permettant de séparer l'ordonnancement de l'application est original et constitue la base de notre proposition d'interface entre la régulation de charge et les applications.

4.1 Introduction

Comme nous l'avons vu dans le chapitre 2 [24, 105], deux classes extrêmes d'ordonnancement applicatif peuvent être distinguées :

- les algorithmes d'ordonnancement dits *statiques* : l'ordonnancement est alors calculé, avant l'exécution du programme, à partir de la connaissance d'une description des processus de l'application et de leurs synchronisations. Si l'on dispose d'un modèle précis de l'architecture, la prise en compte des spécificités de l'exécution permet le calcul d'ordonnancements performants, en particulier lorsque l'on prend en compte les communications.
  La section 4.2 présente trois algorithmes génériques de cette classe qui visent à minimiser respectivement le nombre ou les volumes de communication (Metis, PYRROS), ou les délais induits par les communications (ETF). Chacun de ces algorithmes est dédié à une classe d'applications très spécifique ; ainsi, tout au moins pour la classe restreinte d'applications qui a motivé leur conception, un tel ordonnancement permet d'obtenir de très bonnes performances expérimentales sur des architectures distribuées.
- les algorithmes d'ordonnancement dits *dynamiques*, ou *à la volée*[2] : l'ordonnancement est alors calculé au cours de l'exécution du programme. D'un usage *a priori* très général puisque sans hypothèse sur la structure du programme, ces algorithmes sont très populaires.

Les deux prochaines sections sont consacrées à la présentation de ces principaux algorithmes d'ordonnancement. Aussi bien dans le cadre statique que dans le dynamique, on distingue deux grands types de stratégies qui motivent le calcul de l'ordonnancement :

[2]En anglais : *on-line*

– les stratégies dites de *partage de charge* visent à minimiser l'inactivité des processeurs. Lorsqu'un processeur devient inactif et qu'il existe une tâche prête à être exécutée non encore démarrée, celle-ci lui est affectée. Ainsi, l'objectif d'un algorithme de partage de charge est de maintenir les processeurs occupés par l'exécution de tâches [53, 73].
– les stratégies dites d'*équilibrage de charge* visent à maintenir une charge identique sur chacun des processeurs.

Dans un cadre très général de programmes, les algorithmes de partage de charge (dits aussi algorithmes gloutons ou encore algorithmes de liste) permettent d'obtenir un temps d'exécution très proche de l'optimal, et ce d'autant plus que le programme possède un fort degré de parallélisme. Cette propriété, prouvée théeoriquement pour la première fois par Graham dans [65], explique leur succès en pratique.

Nous rappelons ci-dessous ce résultat fondamental qui motive la grande majorité des stratégies de type partage de charge, aussi bien dynamiques que statiques. Cette propriété est valable pour tout programme applicatif ; sa démonstration est importante car elle peut être étendue à un grand nombre d'algorithmes de partage de charge. Nous reprenons ici la formulation et la preuve donneée dans [80].

Théorème 1 *[65, 80] On considère une machine parallèle constituée de processeurs identiques et partageant un même espace mémoire sans coût de communication.*
Soit T_1 (respectivement T_∞) la durée de l'exécution séquentielle (respectivement parallèle sur un nombre infini de processeurs) d'un programme.
Alors, tout ordonnancement de type partage de charge sur p processeurs a une durée d'exécution T_p bornée par :

$$Max(\frac{T_1}{p}; T_\infty) \leq T_p \leq \frac{T_1}{p} + T_\infty$$

Preuve. Tous les algorithmes d'ordonnancement de type partage de charge contrôlent, au moins implicitement, la gestion des tâches prêtes à être exécutées et non encore démarrées. Supposons ces tâches stockées dans une liste qui évolue dynamiquement. Un processeur, chaque fois qu'il devient inactif, choisit alors dans cette liste la prochaine tâche à exécuter. Si cette liste est vide, le processeur reste inactif jusqu'à ce qu'une tâche soit de nouveau insérée dans la liste.

La preuve présentée ci-dessous consiste à borner les temps d'inactivité des processeurs par le chemin critique du graphe. Soit T_p la durée de l'ordonnancement d'un programme parallèle par un algorithme de liste (gerée de façon centralisée) sur une machine à p processeurs – T_p correspond à la date de fin d'exécution du programme. Soit τ_{j_1} une tâche finissant à la date T_p et d_{j_1} sa date de démarrage. Nous pouvons distinguer deux cas :

Cas 1 : aucun processeur n'a été inactif avant d_{j_1}.

Cas 2 : il existe au moins un processeur inactif à un certain instant avant d_{j_1}.

Dans le second cas, soit θ la plus grande date avant d_{j_1} à laquelle au moins un processeurs est inactif. Puisque τ_{j_1} n'est pas démarrée à l'instant θ sur l'un des processeurs inactifs c'est que τ_{j_1} n'etait pas prête à cette date. Un prédécesseur τ_{j_2} est donc en cours d'exécution à la date θ.

Ce schéma appliqué récursivement – jusqu'à ce que le cas 1 se produise – construit une séquence de tâches $\tau_{j_k} \prec \ldots \prec \tau_{j_2} \prec \tau_{j_1}$ telle que, à tout instant, soit tous les processeurs sont actifs, soit un processeur exécute une tâche τ_{j_i} de cette séquence.

Soit T_1 le temps d'exécution de ce même programme sur une architecture à un seul processeur, c'est-à-dire la charge totale générée par le programme. Le temps d'inactivité I accumulée sur tous les processeurs pendant l'exécution du programme est défini par $I = pT_p - T_1$. Soit $|\tau_{j_i}|$ la durée de la tâche τ_{j_i} ; la durée d'une tâche représente le temps de calcul nécessaire à son exécution. Nous avons ainsi que $I \leq (p-1)\sum_{i=1}^{k}|\tau_{j_i}|$, d'où :

$$pT_p \leq T_1 + (p-1)\sum_{i=1}^{k}|\tau_{j_i}|$$

Soit T_∞ le temps minimal d'exécution d'un programme parallèle sur un architecture avec un nombre non borné de processeurs. T_∞ correspond alors au chemin critique du programme, c'est-à-dire la plus longue séquence de successeurs en terme de somme des temps d'exécution.

Les tâches τ_{j_i}, $1 \leq i \leq k$, étant sur un chemin du graphe, nous avons $T_\infty > \sum_{j=1}^{k}|\tau_{j_i}|$, et donc :

$$T_p \leq \frac{T_1}{p} + \left(1 - \frac{1}{p}\right)T_\infty \tag{4.1}$$

□

Partage versus équilibrage de charge : comparaison théorique. La preuve précédente montre que tout algorithme de type partage de charge permet d'obtenir une exécution efficace, au moins lorsque les communications sont négligeables.

Plus précisément, le nombre de tops d'inactivité dans un algorithme de type partage de charge est majoré par pT_∞, généralement très petit devant le temps séquentiel pour de nombreuses applications fortement parallèles.

Cette remarque est fondamentale car elle permet de montrer que, dans ce cadre

d'applications, le nombre d'opérations d'ordonnancement minimal à réaliser (i.e. création de tâches, placement, préemption, migration) est borné par pT_∞ donc inversement proportionnel au degré de parallèlisme de l'application. Plus le programme est parallèle (i.e. plus T_∞ est petit), moins il y aura d'opérations d'ordonnancement à rèaliser pour garantir l'activité des processeurs [13].
Ce paradoxe apparent est à la base des algorithmes de type vol de travail [13]. Il donne un avantage flagrant aux algorithmes de type partage de charge par rapport aux algorithmes de type équilibrage de charge, qui eux doivent dans tous les cas exécuter un nombre d'opérations d'ordonnancement (notamment migration) au moins égal au nombre de tâches.

4.2 Les stratégies statiques

Les stratégies d'ordonnancement statique sont généralement appliquées avant le démarrage du programme. C'est-à-dire que, indépendemment des données qui lui sont fournies en entrée, le nombre et la taille de tâches qui seront créées peuvent être calculés, de même que la structure des communications et des relations de dépendances $\prec$ entre les tâches. De manière très générale, ces informations sur l'exécution du programme sont représentées sous forme d'un graphe annoté par des coûts, même si la définition précise de ce graphe dépend de l'ordonnancement calculé.

4.2.1 Ordonnancements de type partage de charge

Nous considérons trois algorithmes statiques de type partage de charge ; l'utilisation de connaissances supplémentaires sur le graphe de tâches permet de choisir dans la liste des tâches prêtes celle qui doit être affectée à un processeur inactif. Ainsi, pour des cas spécifiques de graphes de tâches ou de modèles de machine, on peut obtenir des performances supérieures à un ordonnancement glouton [18].

4.2.1.1 Durée des tâches : ordonnancement de Coffman-Graham

Le premier algorithme statique que nous décrivons est celui proposé par Coffman et Graham dans [34]. Cet algorithme prend en entrée un graphe de précédence de tâches unitaires (durée d'exécution == 1) et construit une liste de tâches. Cette liste est triée selon les prioritées calculées pour les tâches en fonction du chemin critique du graphe. L'attribution des priorités est ainsi partie intégrante de l'algorithme de Coffman et Graham :

1. Initialement toutes les tâches du graphe qui n'ont pas de successeur sont étiquetées avec des valeurs arbitraires non dupliquées.
2. Ensuite les tâches qui précèdent les tâche qui viennent d'être étiquetées sont aussi étiquetées par la juxtaposition des étiquettes de ses successeurs dans l'ordre décroissant des étiquettes. Cette étape est répétée jusqu'à l'étiquetage de toutes les tâches du graphe.

La liste de tâches est alors construite, en considerant comme tâche la plus prioritaire celle qui a la plus grande étiquette. Ainsi, la priorité est donnée à la tâche qui a le plus grand nombre de successeurs.

Dans [82] la compétitivité de cet algorithme sur une machine à p processeurs est montrée bornée par $(2 - 2/p)$.

4.2.1.2 Prise en compte des communications : ordonnancement ETF

L'algorithme ETF (*Earliest Task First*) [70] calcule l'ordonnancement des tâches d'un graphe sur une machine distribuée composée de p processeurs identiques reliés les uns aux autres par un réseau de communication sans contention. Ce graphe explicite les relations de dépendance entre les tâches qui correspondent aux communications de données.

Le principe d'ETF est d'obtenir pour chaque tâche la plus petite date de démarrage, dès qu'elle devient prête. Pour cela, les délais de communication engendres par le placement d'une tâche τ sur un processeur p sont considérés ainsi que et la date de disponibilité de p.

Notons que, comme il s'agit d'un ordonnancement statique, la communication d'une donnée produite par une tâche τ_i peut être immédiatement acheminée lors de la terminaison de τ_i. En effet, s'il existe une tâche τ_j successeur de τ_i, elle est déjà placée.

L'application de l'algorithme d'ETF fournit une borne de $\frac{T_1}{p} + (1 - \frac{1}{p})T_\infty + C_{max}$ [85] ; l'analyse est similaire à celle proposée par Graham pour les algorithmes de liste sans communication (cf. théorème 1).

4.2.1.3 Regroupement des communications : ordonnancement DSC

L'algorithme DSC (*Dominant Sequence Clustering*) [129] est un algorithme de regroupement des tâches [106] d'un graphe de précédence. Le principe du regroupement est d'augmenter le grain du programme par la création de groupes (appelés *clusters*) de tâches pour exploiter la localité des données. Chaque groupe représente les tâches allouées à un même processeur d'une architecture à nombre non borné de

ressources de calcul. Cet ordonnancement conduit à éliminer les communications les plus pénalisantes dans le graphe de précédence. Une fois les groupes créés, ils sont ordonnancés sur les p processeurs réels de la machine parallèle cible.

Le résultat immédiat de l'application de cet ordonnancement est la réduction des communications entre les processeurs.

L'algorithme DSC se décompose en autant d'étapes que de tâches dans le graphe de précédence. Son principe est à chaque étape de chercher à réduire le chemin critique – tenant compte des temps de calcul et de communication – en insérant une tâche dans un des groupes déjà créés.

Dans [63] nous retrouvons pour l'algorithme DSC la borne suivante : $(1+\frac{1}{g})T_\infty$, où g représente la granularité du graphe de précédence, c'est-à-dire ici le rapport calcul/communication.

4.2.2 Ordonnancements de type équilibrage de charge

De nombreux problèmes de simulation (notamment en décomposition de domaines) se ramènent à un calcul itératif où, à chaque pas de l'itération, un point de l'espace (généralement en dimension 3) est mis à jour à partir des valeurs, à l'itération précédente, des points qui lui sont voisins. Sur une architecture distribuée, la minimisation du surcoût des communications conduit donc à regrouper les points qui sont voisins de manière à distribuer de manière équitable les groupes entre les processeurs.

D'un point de vue modélisation, un tel schéma de calcul peut être représenté par un graphe de dépendances, non-orienté. Dans ce graphe, la succession de toutes les mises à jour effectuées sur un même point constitue un noeud et les arrêtes correspondent aux dépendances de données entre points voisins. Le graphe est considéré valué : coûts du calcul de la mise à jour pour les nœuds et taille des dépendances de donnée pour les arrêtes.

L'algorithme d'ordonnancement réalise alors le regroupement des noeuds en « grappes » de façon à minimiser les communications entre deux grappes. Pour la construction de ces grappes, il prend en compte aussi les coûts des tâches : l'algorithme essaie de composer des grappes avec des poids de calcul engendrés par les tâches équivalents.

Il existe de nombreux algorithmes de partitionnement de graphe [46] mais très peu se montrent à la fois efficaces et applicables à faible coût. Une classe d'algorithmes particulièrement intéressante est celle des algorithmes de partitionnement multi-niveaux, notamment ceux disponibles dans Metis [78] et SCOTCH [96].

L'idée de base de ces algorithmes multi-niveaux est de transformer le graphe

d'entrée en une représentation « plus petite ». Cette réduction est obtenue à partir d'une fonction de réduction (à faible coût) appliquée iterativement au graphe jusqu'à ce que la taille du graphe, c'est-à-dire le nombre de nœuds, soit suffisament petite. Ce nouveau graphe est alors soumis à un algorithme de partitionnement plus coûteux, générant en sortie les grappes. La fonction inverse de la réduction est applique aussi interactivement en ajustant les frontières des grappes.

Les travaux de Karypis et Kumar [76, 78]. donnent une analyse théorique et pratique des algorithmes multi-niveaux ; cette analyse ne considère que des graphes correspondant à des schémas de calcul itératifs du type de ceux introduits au début de cette section.

4.2.3 Ordonnancement statique et applications régulières

Les stratégies statiques sont donc basées sur le fait de disposer de la connaissance totale (ou partielle) de l'exécution. Dans [105], une telle application est classifiée comme *graphe prévisible*. Les auteurs proposent une mesure, appelée *irrégularité*, définie comme le coût de construction de ce graphe qui contient toute l'information sur l'exécution. Dans tous les cas, l'irrégularité est majorée par le temps d'exécution du programme, puisqu'un traçage de l'exécution permet d'obtenir toute l'information. Dans le cas du calcul d'un ordonnancement statique, on considère que l'irrégularité (i.e. le coût de construction du graphe) est négligeable devant le coût d'exécution ; l'application est alors dite *régulière*.
Moins l'on dispose d'informations sur la trace d'exécution, comme par exemple pour des applications de recherche arborescente [38, 105], plus l'irrégularité est grande ; il est alors nécessaire de recourir à un ordonnancement dynamique, *a priori* sans connaissance sur le futur de l'exécution.

4.3 Les stratégies dynamiques

L'ordonnancement d'un programme *irrégulier* [105] – pour lequel il est impossible de connaître le nombre et la taille des tâches et la structure des communications entre ces tâches avant l'exécution – ne peut être réalisé qu'en cours d'exécution du programme, à la volée, par un algorithme d'ordonnancement dynamique. Un tel algorithme se caractérise par sa réactivité aux évènements du programme au cours de son exécution : par exemple, création ou terminaison d'une tâche, variation des taux d'utilisation des ressources de l'architecture, *etc*. Les algorithmes qui appartiennent à cette classe d'ordonnancement sont couramment appelés algorithmes de *régulation de charge*.

Dans la littérature, de nombreux algorithmes pour la régulation de charge dynamique ont été proposés. Dans les sections 4.3.1 et 4.3.2, nous analysons les caractéristiques particulières de certains de ces algorithmes. En conclusion de cette section, nous étudions les limites théoriques de ce type d'algorithmes.

4.3.1 Équilibrage de charge dynamique

L'équilibrage de charge dynamique est doté d'un mécanisme permettant de distribuer équitablement la charge parmi les processeurs [53, 73]. Si l_i est le taux d'utilisation d'une ressource d'un processeur i, dans une architecture à p processeurs, on veut assurer que $\forall i \in [1,p]$, $\frac{\sum_{j=1}^{p} l_j}{p} - \delta \leq l_i \leq \frac{\sum_{j=1}^{p} l_j}{p} + \delta$, où δ est une marge de tolérance pour la variation de la charge sur un processeur. Pour atteindre ce but, le mécanisme de manipulation des informations de charge est très important dans la majorité des stratégies [127].

Nous présentons ici deux algorithmes d'équilibrage de charge, le premier proposé par M. Willbeek-LeMair et A. Reeves dans [127] et le second par Y. Denneulin dans [38]. Les deux algorithmes ont en commun le déclenchement de leur procédure d'équilibrage de charge, qui est réalisé à l'initiative des processeurs surchargés.

L'algorithme de Willbeek-LeMair et Reeves[3] prévoit un noyau d'exécution non-préemptif et chaque processeur contient un seul flot d'exécution ; dans un tel cas, une tâche démarrée est exécutée jusqu'à sa terminaison et les tâches prêtes non démarrées sont considérées comme aptes à la migration. Le nombre de tâches prêtes, incluant celle en cours d'exécution, reflète l'occupation d'un processeur. Un processeur i est considéré surchargé si l_i est au dessus d'un seuil de surcharge L_{sur} et sous-chargé si l_i est en dessous d'un seuil de sous-charge L_{sous}. Assumons que L_{sur} et L_{sous} sont spécifiés lors d'une étape d'initialisation. Une notion de voisinage est introduite de sorte qu'un processeur connaît la charge de ses voisins plus la sienne – les échanges d'information de charge entre voisins sont réalisés au fur et à mesure de l'avancement du programme.

Le démarrage de la procédure de migration de tâches pour l'équilibrage de charge est initiée sur chaque processeur qui, lors de la réception de l'information de charge d'un voisin, détecte que celui-ci est sous-chargé. Dans ce cas, en se servant de l'information de charge de tous les voisins, la charge du voisinage et la charge idéale pour chaque processeur sont calculées ; si le processeur a un nombre de tâches au dessus de la charge idéale, des tâches sont migrées aux voisins considérés sous-chargés.

Dans l'algorithme de Denneulin, le noyau d'exécution supporte la préemption de tâches et les processeurs peuvent avoir plusieurs flots d'exécution concurrents.

[3] Dans [127] cet algorithme est appelé SID – *Sender Initiated Diffusion*.

Cet algorithme prévoit le démarrage immédiat de chaque tâche devenant prête. L'équilibrage de charge est donc réalisé par la préemption d'une tâche en exécution et sa migration à un processeur sous-chargé. La notion de voisinage n'existe pas dans cet algorithme. Par contre les processeurs sont structurés dans un anneau logique permettant la propagation d'un message contenant la charge d'un processeur. Dans ce schéma, un processeur connaît seulement sa propre charge ; l'information de la charge d'un autre processeur n'est connue qu'au moment où il reçoit le message correspondant.

La décision d'envoyer un message contenant sa charge est prise localement : elle résulte de l'observation d'une variation de la charge du processeur au dessus d'un δ préalablement fourni. L'envoi d'un tel message implique le démarrage d'une procedure d'équilibrage de charge. Une fois envoyé, le message parcourt un par un les processeurs de l'anneau jusqu'à ce que l'un d'eux démarre la procédure de migration. Sinon, le message est enlevé de l'anneau lorsqu'il revient au processeur qui l'a initié. La décision de démarrer une procedure de régulation de charge est prise par le processeur détectant qu'il posséde plus de tâches que le processeur expéditeur du message. Le nombre de tâches nécessaires pour l'équilibre entre les deux processeurs est déterminé en fonction de la moyenne de la charge des deux processeurs. Ensuite les tâches correspondantes sont préemptées puis migrées au processeur sous-chargé.

4.3.2 Partage de charge dynamique

Nous avons vu que tout algorithme de partage de charge permet d'atteindre de bonnes performances lorsque les communications sont de coût négligeable. Ces algorithmes étant très genéraux et ne demandant *a priori* aucune connaissance sur le programme en cours, de nombreux travaux ont cherché à les implanter finement pour minimiser le surcoût introduit pour la gestion des tâches et leur allocation aux processeurs inactifs.

Dans un algorithme de partage de charge, les opérations d'ordonnancement n'ont lieu que lorsqu'un processeur devient inactif, donc en théorie rarement pour un programme très parallèle comme nous l'avons vu en introduction de ce chapitre. Parmi les algorithmes de partage de charge dynamique, les algorithmes de type « vol de travail[4] [13, 16] » sont basés sur cette propriété. Pour limiter le surcoût lié à l'ordonnancement, les opérations les plus coûteuses pour la gestion des tâches sont réalisées non pas à la création (comme c'est le cas habituellement) mais essentiellement lorsqu'un processeur devient inactif. Nous présentons ci-après ce type d'algorithme de partage de charge, qui est intégré dans de nombreuses interfaces de

[4] De l'anglais *work stealing*.

programmation parallèle notamment MultiLisp [69] et Cilk [13].

Dans cet algorithme de partage de charge, une liste L est maintenue avec les tâches prêtes encore non démarrées. La liste L est maintenue de façon distribuée : chaque processeur i, d'une machine parallèle à p processeurs, gère localement un sous-ensemble L_i de cette liste. Ainsi, une tâche qui devient prête sur le processeur i est insérée dans L_i. C'est aussi sur L_i que, à la terminaison d'une tâche, le processeur i cherche une nouvelle tâche à exécuter. Dans le cas où L_i se retrouve vide, une autre sous-liste L_j non vide est choisie pour être « volée ».

Soit t la tâche volée sur le processeur j ; cette tâche avait été créée localement sur j avec un coût minimum. Pour être migrée vers i, il est alors nécessaire de fermer le contexte de t, notamment extraire les données potentiellement dans la pile du processeur i. Autrement dit, construire une clotûre de la tâche t permettant de l'exécuter sur un autre processeur. Cet opération étant coûteuse, elle n'est réalisée que lors du vol de la tâche. Dans les autres cas, la tâche étant exécutée sur le processeur qui l'a créée, elle peut directement utiliser la pile du processeur, sans construire un contexte privé à la tâche. Une manière classique de ne réaliser cette fermeture de contexte que lors du vol d'une tâche est d'utiliser une pile en cactus [114, 13] : la clotûre d'une tâche est alors construite paresseusement en référence à la pile de la tâche qui l'a créée. À partir de cette pile en cactus, il est alors possible, lors d'un vol, de construire le contexte propre de la tâche.

4.3.3 Limites théoriques des stratégies dynamiques

Nous avions vu avec le théorème 1 que tout ordonnancement de type partage de charge (donc même le plus naif sans préemption ni migration) permet d'obtenir de relativement bonnes performances en temps au moins lorsque les communications sont de coût négligeable. Dans cette section, nous étudions si des stratégies plus avancées peuvent améliorer les performances.

Nous montrons tout d'abord que l'introduction d'opérations d'ordonnancement moins elémentaires que le placement et le lancement jusqu'à complétion d'une tâche ne peuvent pas permettre d'obtenir des ordonnancements réellement meilleurs. Puis nous montrons que les stratégies d'ordonnancement aveugles entraînent un surcoût mémoire important. Il est alors nécessaire de considérer des stratégies de partage de charge qui tirent parti d'une connaissance sur le graphe de tâches, soit en le restreignant (c'est le cas de Cilk) soit en l'analysant.

4.3.3.1 Préemption et performances

Les ordonnancements de partage de charge présentés précédemment n'utilisent que des opérations très élémentaires pour ordonnancer les tâches : affectation d'une tâche en attent de traitement à un processeur et exécution – *a priori* non préemptive – de la tâche sur le processeur jusqu'à ce qu'elle se bloque. On peut alors se demander si l'utilisation de la préemption peut permettre d'améliorer cette borne.

Ce problème a été étudié par D. Shmoys, J. Wein et D. Williamson dans [108] ; en généralisant le théorème de Graham, ils ont étudié des bornes inférieures (atteintes) pour le facteur de compétitivité de tout algorithme en ligne même lorsque l'on introduit de la migration ou des tirages aléatoires.

Théorème 2 *[108] Si les temps de communication ne sont pas pris en compte, une borne inférieure pour le facteur de compétitivité d'un algorithme d'ordonnancement déterministe avec ou sans migration, sans connaissance de la durée des tâches avant leur terminaison est* $\left(2 - \frac{1}{p}\right)$.
Un algorithme probabiliste sans préemption ne peut avoir un ratio de performance inférieur à $\left(2 - \frac{1}{\sqrt{p}}\right)$*, ratio qui est obtenu par un algorithme de liste dans lequel les tâches affectées aux processeurs inactifs sont tirées au hasard parmi l'ensemble des tâches prêtes.*

La preuve de la première partie du théorème 2 est reproduite ici dans le but de montrer que la migration de tâches ou l'introduction d'aléatoire ne permet pas d'améliorer ces bornes en comparaison d'un algorithme de liste [80].

Preuve. Pour la preuve, nous prenons un programme parallèle constitué d'une tâche α de durée p et de $p(p-1)$ tâches de durée 1. Pour un tel ensemble de tâches, l'ordonnancement optimal est obtenu en exécutant la tâche α sur un processeur et les autres $p(p-1)$ tâches sur les $(p-1)$ autres processeurs, la durée total de l'exécution est alors p. Pour le même programme, le pire ordonnancement donné une longueur $(p-1)+p$ et est obtenu par l'ordonnancement de la tâche α après les $p(p-1)$ tâches de duration 1. Les tâches étant indifférenciées, sauf par leur durée, même si l'ordonnanceur utilise les opérations de préemption et de migration, un adversaire peut toujours se ramener à ce pire cas, quelles que soient les décisions d'ordonnancement. Le temps $(p-1)+p$ nous donne donc la borne inférieure $\left(2 - \frac{1}{p}\right)$. L'introduction de la préemption ou de la migration de tâches ne permet donc pas d'améliorer ce ratio. □

En conséquence, l'utilisation de préemption ne peut permettre d'améliorer l'ordonnancement.

4.3.3.2 Inefficacité en espace mémoire des stratégies aveugles

Les stratégies dynamiques introduites dans cette section n'utilisent aucune connaissance sur le graphe de tâches. Lorsqu'un processeur devient inactif et qu'il existe plusieurs tâches prêtes, aucune connaissance ne permet donc de déterminer celle qui doit être exécutée en priorité.

Cette absence de connaissance s'avère particulìerement pénalisante en ce qui concerne l'espace mémoire nécessaire [2]. L'exemple ci-dessous, basé sur un ordonnancement en largeur d'un arbre binaire, met en évidence l'inefficacité d'un ordonnancement aveugle.

Considérons une tâche qui crée récursivement deux tâches indépendantes jusqu'à une profondeur de n. Le programme (écrit ici en style ATHAPASCAN-1) est le suivant :

```
void f ( int n ) {
   if (n > 1) {
      fork f( n / 2 ) ;
      fork f( n / 2 ) ;
   }
}
```

Ce programme exécute $\Theta(n)$ opérations. La création récursive des tâches peut être représentée par un arbre binaire équilibré comportant $\Theta(n)$ nœuds, donc de profondeur $\Theta(\log n)$. Tout ordonnancement correspond à un parcours (éventuellement parallèle) de cet arbre.

Ainsi, tout ordonnancement de type partage de charge sur p processeurs (on suppose $p < \frac{n}{\log n}$ conduit à un temps parallèle $O(n/p)$ ce qui est asymptotiquement optimal. Mais l'espace mémoire requis par l'ordonnancement dépend du choix d'affectation des tâches.

Un ordonnancement séquentiel, qui parcourt l'arbre en profondeur d'abord, demande un espace mémoire $\Theta(\log n)$. Dans le pire cas, un ordonnancement à la volée, aveugle sur les dépendances de tâches, effectue un parcours en largeur de l'arbre et requiert donc un espace mémoire $\Theta(n)$. Par contre un ordonnancement qui choisit toujours pour l'affectation à un processeur inactif la tâche la plus profonde requiert un espace mémoire beaucoup plus petit, majoré par $\Theta(p \log n)$.

Ainsi, sans connaissance supplémentaire sur les tâches à ordonnancer, aucune borne raisonnable ne peut être donnée sur l'espace mémoire requis par une exécution parallèle.

Dans [15], R. Blumofe et C. Leiserson montrent qu'il existe des programmes parallèles pour lesquels aucune accélération significative ne peut être obtenue sans explosion mémoire. Deux alternatives peuvent être alors considérées :

– Restreindre le parallélisme pouvant être décrit pour permettre des ordonnancements assurant un espace mémoire borné.
  C'est le choix effectué dans le langage Cilk [13] où seuls des programmes de type série-parallèle[5] peuvent être écrits.
– Calculer l'ordonnancement à partir d'informations additives, pour éviter un calcul complètement aveugle. La connaissance d'un ordre séquentiel d'exécution valide permet de borner l'espace mémoire relativement à celui pris par cette exécution séquentielle [91].
  C'est le choix effectué dans le langage ATHAPASCAN-1 ; d'une part un ordre séquentiel implicite peut être utilisé pour le calcul de l'ordonnancement [59]. D'autre part, des informations additives peuvent être fournies, comme des priorités par exemple, pour spécialiser l'ordonnancement.

Dans ces deux alternatives, l'ordonnancement, même si il est calculé à la volée lors de l'exécution, utilise donc des informations additives sur le programme qui sont nécessaires pour garantir un espace mémoire borné. L'ordonnancement est alors *hybride* puisque, bien que calculé dynamiquement, il utilise, comme un ordonnancement statique, des informations sur le futur de l'exécution.

4.4 Ordonnancement hybride

Les algorithmes d'ordonnancement statiques ou dynamiques vus précédemment correspondent à deux cadres extrêmes dans lesquels de nombreuses applications pratiques ne s'inscrivent pas :

– d'une part, l'ordonnancement ne peut pas être calculé statiquement sauf si l'on a préalablement exécuté ; en effet, du fait des branchements conditionnels non-prédictibles, le graphe de dépendance de l'application n'est que partiellement connu. Il peut en outre exister une forte imprécision sur les durées des tâches : sur beaucoup de processeurs, le temps d'une opération dépend largement du contexte d'exécution (opérandes présents ou non dans le cache, *etc*.) et est donc souvent difficile à estimer.
– d'autre part, un ordonnancement dynamique glouton, qui ordonnance les processus de manière aveugle sans prise en compte de leurs relations, conduit à des performances faibles ; en effet, le coût des communications ne peut pas être raisonnablement négligé sur les architectures distribuées disponibles. Et même pour des algorithmes permettant de limiter les communications (c'est le cas par exemple des algorithmes récursifs qui découpent le problème en

[5] *fork-join* emboîtés

sous-tâches indépendantes [74]), l'espace mémoire requis par un tel ordonnancement s'avère prohibitif [13, 59].

Considérons par exemple le cas élémentaire du produit de deux matrices carrées ; l'utilisation d'un ordonnancement dynamique glouton (donc aveugle) sans prise en compte des coûts de communication conduit alors à des performances désastreuses. Pourtant, pour ce problème sur une architecture constituée de processeurs identiques, un ordonnancement statique bloc-cyclique conduit à de très bonnes performances.

C'est pourquoi en pratique, l'algorithme d'ordonnancement applicatif est souvent hybride. L'application s'exécute comme une succession (séquentielle) de phases [105] ; au début de chaque phase, on a la connaissance de ce qui va se passer dans le futur proche de l'exécution (la phase suivante), mais aucune (ou peu d') informations sur les phases ultérieures. L'ordonnancement est alors calculé dynamiquement (en cours d'exécution) : différents algorithmes d'ordonnancement (statiques ou à la volée) peuvent être utilisés pour chacune des phases. Le projet inter-PRC Stratagème a montré que de nombreuses applications pratiques se situaient dans ce cadre [105].

4.5 Fondements communs : l'analyse du flot de données

Pour permettre à une application d'être exécutée avec un tel algorithme hybride (*a priori* quelconque et pouvant varier d'une phase à l'autre), il est nécessaire de séparer l'algorithme d'ordonnancement du programme applicatif dans deux modules distincts, en définissant leurs interactions.

Le point commun entre les différents algorithmes d'ordonnancement presentés dans ce chapitre est d'être basés sur une analyse plus ou moins partielle de la trace d'exécution du programme. De manière générale cette trace peut être modélisée par un graphe de flots de données, qui décrit les entités à placer et ordonnancer : les données et les tâches. Ce graphe peut être annoté par des informations, selon l'algorithme d'ordonnancement considéré.

Nous proposons dans ce chapitre que les opérateurs permettant de construire et manipuler ce graphe soient au centre de l'interface entre le module ordonnancement et le module applicatif.

La section 4.5.1 définit le graphe de flot de données. Le problème de l'ordonnancement d'un graphe de flot de données est formalisé dans la section 4.5.2.

Dans la conclusion de ce chapitre, section 4.6, nous identifions les relations entre l'ordonnancement et le graphe de flot de données.

4.5.1 Représentation de l'exécution par un graphe

L'exécution d'une application parallèle peut être décrite comme une succession d'opérations (ou tâches) exécutées par des tâches sur des données partagées. Nous supposons dans la suite que le programmeur fourni la description de cette sucession d'opérations pour une application de façon à permettre la construction d'un graphe décrivant activités du programme [57]. Selon la nature de l'application, la construction de ce graphe peut être réalisée (*i*) avant le démarrage de l'exécution, soit par des techniques de compilation comme dans HPF [97] ou soit par des outils de description explicite de graphes comme PYRROS [130], ou (*ii*) seulement à l'exécution, à l'aide d'un mécanisme d'interpretation abstraite, plus ou moins explicite, supporté par un environnement d'exécution, comme dans Jade [102], Cilk [14, 13] ou ATHAPASCAN-1 [58].

Un tel graphe contient des informations sur le programme qui peuvent être exploitées par l'algorithme de régulation de charge. En particulier il donne les précédences entre les calculs, mettant en évidence la concurrence des calculs et les point de synchronisation ; il fournit ainsi une vision plus ou moins lointaine du future de l'exécution. Le concept de graphe décrivant une application est généralement utilisé pour l'implantation des stratégies statiques [37] ; les algorithme d'ordonnancement dynamique manipulent généralement un graphe décrivant le futur proche (les tâches prêtes) de l'exécution.

Program 4.1 Calcul d'une integrale par Newton-Cotes. Calcul par découpe récursive de l'integrale d'une fonction `foo` sur l'intervalle $[a, b]$ par la méthode de Newton-Cotes.

```
 1: double Calcul( double a, double b )
 2: {
 3:   if( b-a < h )
 4:     return foo(a,b) ;
 5:   else {
 6:     double res1, res2 ;
 7:     res1 = Calcul( a, (a+b)/2 ) ;
 8:     res2 = Calcul( (a+b)/2, b ) ;
 9:     return( res1 + res2 ) ;
10:   }
11: }
```

Ainsi, un critère de classification plus déterminant pour les stratégies de régulation de charge est la connaissance qu'un algorithme dispose sur le graphe décrivant l'application [105]. Une application peut être régulier ou irrégulier : une applica-

tion régulier est celle dont la structure de l'exécution peut être facilement calculé à l'avance ; une application irrégulière, au contraire, la structure de l'exécution est très coûteuse à connaître à l'avance – voir impossible avant la fin de l'exécution –. Néanmoins, en pratique, un grand nombre d'applications sont *semi-régulières*, par exemple dynamique moléculaire [7] et Cholesky creux [43].

Dans la suite nous caractérisons deux représentations d'un programme sous la forme d'un graphe orienté sans cycle. Nous utilisons le programme 4.1 pour donner un exemple de ces deux types de graphes. Nous considerons que ces représentations sont une spécialisation d'un graphe plus général : le graphe de tâches. Un graphe de tâches est composé des nœuds représentant les tâches et des arrêtes non orientées représentant les dépendances entre les tâches.

Le graphe de précédence. Un graphe de précédence définit un ordre partiel entre les tâches selon la sémantique spécifiée par le langage. Cet ordre traduit les relations de dépendance entre les tâches comme des spécifications d'une ordre de précédence : les arcs entre les nœuds sont orientés.

Un graphe de précédence est donné en exemple dans la figure 4.1, où les tâches sont représentées par des cercles et les relations de précédence par des arcs. Les dépendances étant introduites pour régler des conflits d'accès à des données, elles peuvent être interprétées comme des communications entre les tâches. Dans ce type de graphe une tâche est considérée prête dès que toutes les tâches qui la précéde dans le graphe sont terminées.

Les graphes de précédence peuvent être annotés. Dans ce cas, des informations additionnelles sont disponibles au niveau de chaque élément (nœud ou arrête) du graphe. Par exemple le coût d'exécution de chaque tâche et le coût d'une communication entre deux tâches.

Notons que les graphes du type série-parallèle *fork/join* sont des graphes de précédence.

Le graphe de flot de données. Si les dépendances entre tâches sont considérées comme des communications d'échange de données, il est possible d'ajouter les informations sur les données communiquées dans le graphe : ce graphe est appelé alors graphe de flot de données. Un graphe de flot de données est un graphe biparti, où sont représentées les tâches est les données. La figure 4.2 représente le graphe de flot de données pour une exécution du programme 4.1.

La différence entre les graphes de précédence et de flot de données se situe au niveau des synchronisations. Dans les graphe de flot de données une tâche est

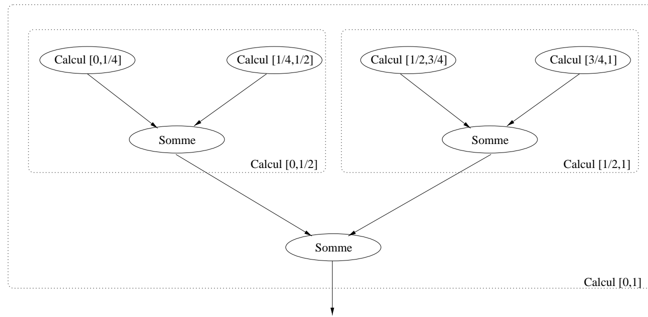

Figure 4.1 *Un graphe de précédence pour le programme 4.1. Les tâches sont représentées par des cercles. Dans ce type de graphe, une tâche est considéré prête dès que toutes les tâches prédécesseurs ont été finies.*

considérée prête dès que les données qu'elle a en entrée sont disponibles. Notons néanmoins qu'un graphe de flot de données contient les informations d'un graphe de précédence.

Pour la régulation de charge ce type de graphe fournit plus d'informations, puisque les données communiquées entre les tâches sont identifiées – les tailles des données peuvent également être fournies –. Cette information peut être utilisée pour exploiter la localité des données lors du placement de tâches ou pour le routage des données accédées vers le site d'exécution d'une tâche.

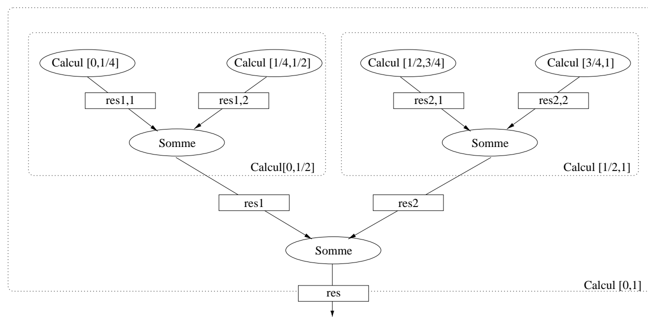

Figure 4.2 *Un graphe de flot de données pour le programme 4.1. Les tâches sont représentées par des cercles et les données par de rectangles. Dans ce type de graphe, une tâche est considéré prête dès que toutes les données prédécesseures ont été produites.*

Dans cette thèse nous avons choisi de représenter un programme parallèle par un graphe de flot de données. Ainsi, dans la section suivante nous abordons le problème de l'ordonnancement d'un graphe de flot de données sur une architecture parallèle cible.

4.5.2 Formalisation du problème d'ordonnancement

La régulation de charge traitée dans cette thèse considère comme modèle de machine parallèle [60] une architecture composé d'un ensemble $\mathcal{P} = P_1, P_2, \ldots, P_p$ de processeurs ; chaque processeur P_i possède son propre module de mémoire M_i et accède une mémoire partagée qui sert de support de communication entre les processeurs. Le modèle de programme parallèle est composé par un ensemble $\mathcal{T}$ de tâches prenant en entrée une donnée x sur un ensemble $\mathcal{X}$, et qui produit un ensemble $\mathcal{O} = \{(\tau_1, x_1), (\tau_2, x_2), \ldots, (\tau_n, x_n)\}$, telle que τ_i est une tâche qui s'exécute avec la donnée x_i en entrée.

Nous considérons que l'exécution d'un couple $(\tau_i, x_i) \in \mathcal{O}$ est conditionnée par des contraintes de dépendance $\prec$, puisque son entrée x_i peut être le resultat du calcul d'un autre couple (τ_j, x_j). Ainsi l'ensemble $\mathcal{O}$ définit un graphe de flot de données G. La taille de ce graphe est notée $\#G$ et représente le nombre de tâches qu'il contient.

Définition 1 *Un* **graphe de flot de données** G *est la représentation d'une exécution d'un programme parallèle :* $G = (V, E)$*. Les nœuds de* G *sont formés par* $V = \mathcal{T} \cup \mathcal{X}$*, c'est-à-dire par l'ensemble de tâches et des données manipulées par le programme, et les arrêtes* E *par les accès des tâches aux données. Le graphe* G *est dit orienté sans cycle et biparti :* $E = (\mathcal{T} \times \mathcal{X}) \cup (\mathcal{X} \times \mathcal{T})$.

Nous rajoutons à la définition de graphe de flot de données la notion de graphe annoté. Un graphe de flot de données G est dit *annoté* si ses nœuds sont valués par des poids. Ainsi, dans le cas du graphe biparti décrivant tâches et données, les poids peuvent représenter le coût d'exécution d'une tâche ou la taille de la donnée.

Dans G les arrêtes traduisent les liens de dépendance entre les tâches et les données. Ainsi une arrête $(\tau, x) \in E$ implique un droit d'écriture de la donnée x par la tâche τ. Symétriquement, une arrête $(x, \tau) \in E$ implique le droit de lecture de la donnée x par la tâche τ. Un ensemble d'états est associé aux nœuds de G pour déterminer les contraintes de dépendances [59]. Nous résumons ces états dans le tableau 4.1.

Il nous reste ainsi à formaliser le comportement de l'ordonnancement sur un graphe de flot de données.

Définition 2 *Dans ce formalisme,* **l'ordonnancement est une fonction** $\mathcal{S}$ *qui*

Tableau 4.1 *États des nœuds d'un graphe de flot de données. Nous présentons d'une manière informelle les états des nœuds d'un graphe. La définition formelle de ces états se trouve dans [59].*

État	Tâche	Donnée
Attente	Au moins une donnée en entrée de la tâche n'est pas prête.	Au moins une tâche qui est en cours d'exécution posséde le droit d'écriture sur la donnée.
Prêt	Toutes les données en entrée de la tâche sont prêtes.	Aucune tâche qui ne soit pas terminée ne possede le droit d'écriture sur la donnée.
Exécution	La tâche est en cours d'exécution.	—
Bloqué	L'exécution de la tâche a été interrompue.	—
Terminé	La tâche a fini son exécution.	Aucune tâche ne possède plus de droits d'accès sur la donnée.

prend en entrée une architecture cible $\mathcal{P} = P \cup M$, $P = \{P_1, P_2, \ldots, P_p\}$ et $M = \{M_1, M_2, \ldots, M_p\}$, et un graphe de flot de données G composé de $\#G$ tâches τ_i, $1 \leq i \leq \#G$, chaque tâche de durée $|\tau_i|$. Cette fonction est appliquée sur chaque élément de G, tâche ou donnée, de façon à lui attribuer un site, processeur ou mémoire, et une date, de démarrage d_{τ_i} ou d'allocation d_{x_i}.

$$\mathcal{S} : G \longmapsto \mathcal{P} \times \mathbb{R}^+$$

$$\begin{cases} x_i & \longmapsto (M_j, d_{x_i}) \\ \tau_i & \longmapsto (P_j, d_{\tau_i}) \end{cases}, d_{x_i} \leq d_{\tau_i}$$

$\mathcal{S}$ fournit en sortie $T_p(G)$ une exécution de G sur les p processeurs de $\mathcal{P}$ qui vérifie $d_{\tau_i} \geq$ Prêt(τ_i), où Prêt(τ_i) est la date à laquelle la tâche τ_i devient prête.

Nous considérons dans la définition 2 tant le cas où G est produit *avant* le démarrage du programme que le cas où G est produit en cours d'exécution. La principale différence est que dans le premier cas, la fonction d'ordonnancement est appliquée dès que le graphe est entièrement décrit, alors que dans le deuxième, la fonction d'ordonnancement est appliquée lors de chaque changement du graphe.

Une fois que nous avons défini un graphe de flot de données et le comportement de l'ordonnancement face à ce graphe, nous pouvons définir une interface des services entre ces deux niveaux.

4.6 Le constructeur de tâches : l'interface entre l'application et l'ordonnancement

L'ordonnancement étant appliqué sur le graphe, il est nécessaire de définir : (*i*) l'accès au graphe, permettant son exploration et (*ii*) la réactivité de l'algorithme d'ordonnancement aux évènements survenant dans le graphe. Un module sert d'interface entre l'application et le noyau d'ordonnancement : le « constructeur de tâches » (MCT, module de construction de tâches, pour abréger). Le MCT (appelé *Job Buider* dans [27]) produit des descriptions de tâches en liaison avec un programme en exécution dans un niveau applicatif. Dans le cas où sont décrites des relations de dépendance entre les tâches un graphe est construit et géré par le MCT.

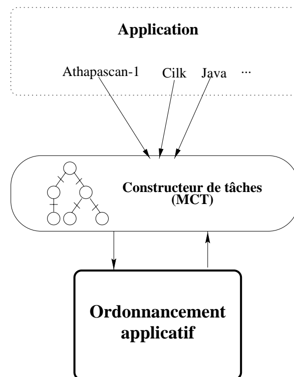

Figure 4.3 *Le constructeur de tâches : l'interface entre l'application et l'ordonnancement. Le module de construction de tâches (MCT) est responsable pour créer la description des tâches produites par un programme en exécution et de construire un graphe de flot de données à partir des dépendances entre les tâches.*

La figure 4.3 représente le MCT comme l'interface entre le noyau d'ordonnancement et l'application. Les services nécessaires aux échanges sont présentés ci-dessous.

Parcours du graphe de flot de données. Toutes les informations nécessaires à l'algorithme d'ordonnancement se trouvent dans le graphe de flot de données maintenu au niveau de l'application. Le niveau ordonnancement doit être capable d'obtenir ces informations. Il doit, à partir d'une tâche ou d'une donnée de ce graphe, pouvoir avancer ou reculer dans la description. Comme il s'agit d'un graphe

biparti – tâches et données – il est nécessaire de distinguer si l'élément de départ de l'exploration est une tâche ou une donnée :

- *À partir d'une tâche* : les éléments qui précèdent et suivent une tâche dans le graphe sont des données, respectivement en lecture et en écriture. Le MCT doit donc être capable d'informer quelles sont les données en lecture ou écriture d'une tâche.
- *À partir d'une donnée* : les éléments qui précèdent et suivent une donnée dans le graphe sont des tâches. Ces tâches sont respectivement la tâche qui l'écrit et celle qui la lit. Le MCT doit donc être capable d'informer quelle est la tâche qui écrit ou lit une donnée.

Il est à noter que, si le graphe est annoté, chacun de ses élément doit permettre à l'algorithme d'ordonnancement d'accéder à son information de coût.

Réactivité de l'ordonnancement. La réactivité d'un algorithme d'ordonnancement est relative au délai entre le moment où un evenement de régulation est déclenché et celui où la décision de la régulation associée est réalisée. Dans le cas d'un algorithme d'ordonnancement statique, son application est liée à la disponibilité du graphe décrivant le programme parallèle. Dans le cas d'un algorithme d'ordonnancement appliqué sur un programme produisant le graphe à la volé le déclenchement de l'ordonnancement se fait nécessairement en réponse à des changements dans le graphe. Or ces changements sont observés lors de l'insertion de nouveaux nœuds ou lors du changement de l'état d'un des nœuds. L'ordonnancement doit ainsi être *actionné* lors d'une modification du graphe et immédiatement initier son traitement.

4.7 Conclusion

Dans ce chapitre nous avons présenté une discussion sur les algorithmes d'ordonnancement et leur relations avec un programme parallèle.

Nous avons abordé les principes des algorithmes dynamiques et statiques pour ensuite identifier leur interprétation des programmes parallèles. Nous avons identifié que la représentation d'un programme par un graphe de flot de données est assez générale : elle permet à la fois la description du parallélisme de l'application et son exploitation par un noyau d'ordonnancement.

Nous avons aussi identifié dans la section 4.3.3.2 qu'il n'est pas possible, dans un cadre général, d'ordonnancer un programme parallèle sans une connaissance sur le futur, au risque d'exploser en mémoire.

Finalement, dans la section 4.6 nous avons caractérisé le fonctionnement d'un algorithme d'ordonnancement sur un graphe de flot de données et nous avons aussi identifié les échanges réalisés entre l'ordonnancement et le graphe de flot de données et caractérisé un module dédié à la construction et à la manutention d'un graphe de flot de données représentant un programme en exécution.

5

L'exécutif : interface entre l'ordonnancement et l'architecture

Dans ce chapitre, nous analysons les opérations réalisées sur une machine parallèle, lors de la régulation d'un programme, de manière à identifier les besoins d'un noyau d'ordonnancement face aux ressources de l'architecture. Cette analyse aboutit à la caractérisation d'une interface entre l'ordonnancement applicatif et une architecture de type réseau de stations SMP : l'exécutif.

Dans la section 5.1 nous présentons les outils de base permettant la programmation des architectures parallèles. Le modèle d'une machine abstraite est introduit dans la section 5.2 ; l'ordonnancement applicatif est implanté sur cette machine abstraite. Les mécanismes employés par l'ordonnancement applicatif, notamment ceux qui pour la manipulation de données et de tâches, sont présentés dans la section 5.3. La section 5.4, présente l'exécutif : l'interface entre le noyau exécutif et le niveau d'ordonnancement.

5.1 Outils de base pour la programmation des architectures parallèles

Cette section présente une classification générale des architectures parallèles (section 5.1.1) et les outils de base permettant leur programmation (sections 5.1.2

et 5.1.3).

5.1.1 Différents types d'architecture parallèle

Trois types d'architecture peuvent être distingués : des architectures à mémoire partagée, des architectures parallèles et des architectures distribuées[1].

Les « architectures à mémoire partagée » sont de plus en plus présentes comme outils de travail. Les machines couramment appelées SMP – multiprocesseurs symétriques[2] –, appartiennent à cette classe d'architecture. Les SMP disposent d'au moins deux processeurs qui partagent une mémoire commune. Mise à part la contention dans l'accès à la mémoire, il n'existe aucune forme de synchronisation implicite entre les processeurs au niveau de l'architecture ; d'autres formes de synchronisations peuvent être introduites explicitement par les programmes en exécution, par l'utilisation d'un verrou ou de mécanismes d'échange de messages, par exemple. Les machines SMP les plus performantes d'aujourd'hui comptent quelques dizaines de processeurs.

La dénomination « architecture parallèle » est généralement employée pour designer les ordinateurs conçus pour être de puissantes ressources de calcul. Ces architectures sont dotées de plusieurs *nœuds* de calcul, chacun disposant d'au moins un processeur (un nœud peut être un SMP) et d'une mémoire locale. Les nœuds ne partagent pas de mémoire commune : un réseau permet les échanges de messages entre eux. Généralement tous les nœuds d'une machine parallèle sont identiques.

Une « architecture distribuée » est dotée de nœuds de calcul disposant d'au moins un processeur et d'une mémoire locale. Comme dans les architectures parallèles, il n'existe pas de mémoire commune entre les nœuds et les échanges des données entre les nœuds sont aussi réalisés par des messages sur un réseau. Cependant, les architectures distribuées diffèrent des architectures parallèles car elles sont composées de nœuds conçus pour être des ordinateurs à l'usage autonome (*a priori* des nœuds non identiques), et d'autre part d'un réseau d'interconnexion supportant un débit de transfert normalement moins élevé.

À cause de l'évolution technologique, il est difficile de déterminer avec exactitude les limites qui définissent une architecture parallèle et une architecture distribuée [71]. Cependant, d'après son organisation, une architecture distribuée peut être considérée comme un réseau de stations de travail, un NOW, ou comme une grappe de stations de travail, un COW[3]. Une COW est supposée dédiée à une application,

[1] Nous renvoyons le lecteur intéressé aux travaux de Flynn [51], de Tanenbaum [118] et de Steen et Dongarra [123] pour plus d'informations sur les taxonomies des architectures parallèles.

[2] De l'anglais *symetric multiprocessors*.

[3] De l'anglais, respectivement : *network* et *cluster of workstations*.

tandis qu'un NOW est supposé partagé entre plusieurs utilisateurs ou applications.

Dans un souci de simplification, nous dénommons *machine à architecture parallèle*, ou simplement *machine parallèle*, n'importe quelle configuration d'ordinateur qui présente les caractéristiques d'un des trois cas cités ci-dessus. Nous employons aussi le terme nœud pour faire référence à un élément dissociable de cette machine parallèle : un nœud est composé d'un espace mémoire et d'un ensemble de processeurs.

5.1.2 Exploitation du parallélisme inter-nœuds et communications

Pour exploiter le parallélisme, les couches de programmation de base permettent de créer des processus et de les synchroniser.

Même si certains systèmes permettent de créer des processus sur des nœuds distants (par exemple *spawn* en PVM [62]), la création de processus est généralement une opération locale à un nœud (*fork* en Unix). La création à distance est alors réalisée par des primitives de communication en exploitant un réseau de communication. La synchronisation entre les processus est réalisée aussi sur ce réseau. Cela permet les échanges d'informations entre les processus par différentes primitives :

– Communication : *socket*, PVM [62], MPI [110].
– Accès à un module de mémoire distante (*remote put* et *get*) : MPI-2 [88].
– Appel de procédure à distance : RPC [8].

5.1.3 Recouvrement des temps d'attente par le *multithreading*

Pour pouvoir recouvrir les temps d'attente dus aux synchronisations entre les processus en exécution, la technique la plus courante est l'utilisation de *multithreading* (cf. section 2.2.1). Cette technique implique de diviser le flot d'exécution d'un processus en divers flots indépendants, ou processus légers, chacun responsable de la réalisation une partie du travail. Lorsqu'un de ces flots exécute une opération de synchronisation engendrée, par exemple, par une prise d'un verrou ou par une communication, le flot est bloqué et le processeur libéré pour l'exécution d'un autre flot. Ce concept de processus légers coopérants est assez ancien [40], mais il a évolué lors de l'émergence d'Unix : un processus léger exécute un flot d'instructions séquentiel sur un espace d'adressage (une mémoire) partagé avec des autres processus

légers[4].

Généralement la gestion des processus légers est réalisée par une couche logicielle, comme c'est le cas de POSIX [1]. Un processus léger est alors représenté par une pile de données et l'état des registres. Toutes les autres informations nécessaires (le code à exécuter, les données globales, les fichiers, *etc.*) sont manipulées par effet de bord au niveau du processus[5] et partagées par tous les processus légers. La gestion des conflits d'accès à cette mémoire est rendue possible par des outils de synchronisation, les plus classiques étant les verrous, sémaphores et les variables de condition.

En outre, le *multithreading* permet une exploitation immédiate du parallélisme dans les architectures SMP.

Pour avoir plus de détails sur les processus légers nous renvoyons le lecteur au travail de A. Carissimi [22].

5.2 Une abstraction de machine parallèle

Avec l'ensemble d'outils présentés dans la précédente section, il est possible de construire une machine parallèle abstraite (figure 5.1).

Cette machine abstraite est constituée d'un ensemble de nœuds de calcul. Un nœud de cette machine abstraite est implanté par un processus (lourd). La communication entre les nœuds est possible grâce à un réseau complet. Chacun de ces nœuds fournit des ressources d'exécution, mémoire et calcul, pour l'exécution (parallèle ou concurrente) de plusieurs processus légers, sur lesquels sont exécutés les tâches de l'application. Ces processus légers sont considérés, en quelque sorte, comme les *processeurs virtuels* de la machine abstraite.

Un point à souligner qui dans cette machine abstraite, les caractéristiques d'hétérogénéité (dans la configuration des nœuds ou du réseau de communication) peuvent être prises en compte par l'algorithme de régulation.

Dans la suite de ce document nous exploitons ce modèle de machine parallèle pour l'opération de l'ordonnancement applicatif.

[4]Dans la terminologie Unix, un processus léger correspond à un *thread* et un nœud correspond à un processus lourd.

[5]D'où la distinction entre processus « léger » et processus « lourd ».

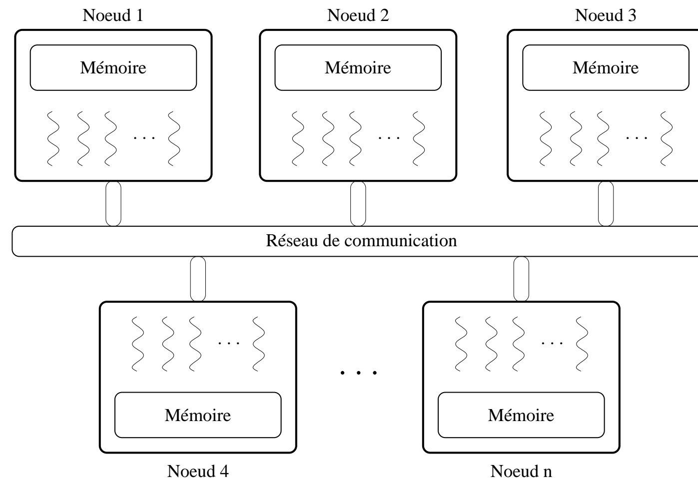

Figure 5.1 *Représentation d'une machine parallèle abstraite. Une machine parallèle est composée d'un ensemble de nœuds de calcul, chaque nœud dispose d'une mémoire et est capable de supporter l'exécution concurrente de processus légers. Un réseau de communication permet la coopération entre les nœuds.*

5.3 Mécanismes pour l'ordonnancement applicatif

Cette section présente les opérations nécessaires à la machine abstraite présentée dans la section précédente pour réaliser l'ordonnancement applicatif. Les services considérés sont ceux employés pour le transfert de tâches et de données entre les nœuds (cf. section 2.2) et pour la manipulation des processus légers.

5.3.1 Manipulation des données

Dans un environnement d'exécution à mémoire partagée, se retrouve une série de problèmes liés à la gestion de données dans une structure hiérarchique de mémoire – c'est-à-dire une hiérarchie composée, par exemple, de mémoire (*i*) disque, (*ii*) principale et (*iii*) cache. D'une manière générale, la gestion de données implique le contrôle des transferts successifs entre ces niveaux, en utilisant des mécanismes implantés sur l'architecture de la machine [118].

Dans des architectures à mémoire distribuée, un nouveau niveau de mémoire doit être considéré, celui de la localisation de la donnée. Donc un ensemble de services doit être fourni pour permettre à l'ordonnancement applicatif de faire le

cheminement des données : le *placement* et la *migration*. Le placement consiste à attribuer un espace mémoire à une donnée sur le module de mémoire d'un des nœuds de la machine virtuelle, et la migration consiste à enlever une donnée d'un module pour le placer sur un autre module.

5.3.2 Entrelacement de tâches

D'une façon semblable aux données, les tâches doivent elles aussi être placées, et éventuellement migrées, sur un nœud. Cependant elles doivent aussi être exécutées, donc démarrées sur un flot d'exécution. Ainsi, les opérations nécessaires à l'ordonnancement applicatif d'une architecture comprennent celles destinées aux communications, permettant de les placer et de migrer les tâches, et aussi celles destinées au contrôle des flots d'exécution. Ces dernières sont concentrées sur un module nommé *pool d'exécution*.

Un pool d'exécution consiste en un ensemble de processus légers qui partagent l'accès aux processeurs et à la mémoire d'un nœud. L'ordonnancement de ces processus légers est supposé implanté par un niveau d'ordonnancement système, qui considère tous les processus légers identiques. Chaque processeur léger est considère comme un processeur virtuel par le noyau d'ordonnancement applicatif, capable d'exécuter séquentiellement les instructions d'une tâche. À la fin d'une tâche il est prêt à en exécuter une nouvelle.

Le nombre de processeurs virtuels d'un pool d'exécution est variable. Ils sont créés et détruits selon les décisions de l'ordonnancement applicatif. De plus, l'ordonnancement applicatif peut démarrer sur un de ces processeurs virtuels une tâche et éventuellement la préempter pour la redémarrer *a posteriori*, sur le même nœud ou sur un autre.

5.4 L'exécutif : l'interface entre l'ordonnancement et l'architecture

L'ensemble des opérations présentées dans la section précédente offre les ressources nécessaires à la définition d'un module permettant la création et la gestion d'une machine abstraite (section 5.2, figure 5.1). Ce module est appelé « exécutif ».

Les services utilisés par l'ordonnancement applicatif pour créer et gérer une machine abstraite sont listées ci-dessous. Dans la prochaine section sont présentés les services qui l'exécutif doit offrir pour permettre le contrôle de l'exécution des programmes parallèles. Ainsi, l'exécutif :

– offre les moyens de création de nœuds, à partir de processus lourds, et garanti l'accès aux ressources de calcul du nœud physique d'une machine réelle.
– doit permettre la création des processus légers à l'intérieur des nœuds et garantir leur accès aux processeurs d'un nœud physique de la machine réelle.
– maintient un réseau de communications entre les nœuds pour permettre la synchronisation (et les communications) entre les nœuds.
– L'exécutif offre des ressources pour la synchronisation entre les processus légers à l'interieur d'un nœud.

5.4.1 L'interface de l'exécutif

Pour que le noyau d'ordonnancement applicatif puisse exécuter un programme, l'exécutif offre un ensemble de services permettant le contrôle de l'exécution de tâches et de la localisation de données. Ces services sont identifiés dans la figure 5.2 et discutés ci-dessous.

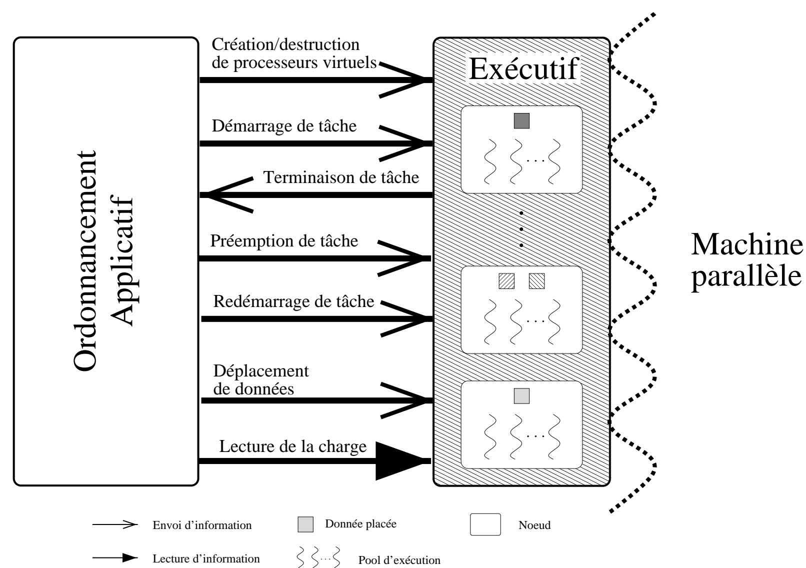

Figure 5.2 *L'exécutif : l'interface d'accès aux ressources de la machine parallèle. Représentation du module responsable de la gestion de la machine réelle et de son interface de services.*

Création et destruction des processeurs virtuels. Le nombre de processeurs virtuels dans un pool d'exécution définit le nombre de flots d'exécution concurrents dans un nœud. Ainsi, selon les décisions prises par l'ordonnancement applicatif, des processeurs virtuels peuvent être ajoutés ou enlevés du pool d'exécution d'un nœud en fonction des ressources physiques du nœud et des besoins du programme en exécution.

Démarrage de tâches. L'opération de démarrage consiste à attribuer la responsabilité de l'exécution d'une tâche à un processeur virtuel. Ce processeur virtuel est choisi parmi l'ensemble des processeurs virtuels inactifs dans la machine abstraite.

Terminaison de tâches. Pour pouvoir avoir le contrôle des processeurs virtuels en train d'exécuter une tâche et de ceux qui sont inactifs, l'ordonnancement applicatif doit être informé à chaque terminaison de tâche.

Préemption de tâches. Par une décision d'ordonnancement applicatif, l'exécution d'une tâche peut être préemptée, c'est-à-dire que la tâche peut être suspendue pendant une certaine période de temps et son processeur virtuel libéré.

Redémarrage de tâches. Le redémarrage est une opération similaire au démarrage, cependant l'exécution d'une tâche est reprise à partir de l'instruction où elle a été bloquée. L'ordonnancement applicatif peut se servir des opérations de préemption et de redémarrage pour réaliser la migration de tâches en cours d'exécution.

Déplacement de données. Les opérations de déplacement de données permettent de placer, et éventuellement de migrer, les données manipulées par l'application sur un module de mémoire d'un nœud.

Informations de charge. Ce service permet la lecture des informations de charge de la machine.

5.5 Conclusion

Nous avons abordé dans ce chapitre la construction d'un module sur lequel l'ordonnancement applicatif est capable de créer une machine abstraite et, sur cette machine abstraite, de contrôler l'exécution d'un programme parallèle.

Nous avons aussi identifié l'interface de ce module, dénommé « exécutif » avec un noyau d'ordonnancement applicatif.

L'exécutif est traité dans la suite de ce document comme l'abstraction du niveau d'architecture dans un environnement d'exécution parallèle.

6

Proposition d'un noyau générique pour l'ordonnancement

6.1 Introduction

Une stratégie d'ordonnancement est construite à partir de deux blocs de base (cf. chapitre 2) : le sous-système de décision et le sous-système d'inspection de charge. Cette structuration permet la définition et la construction de diverses politiques d'ordonnancement. Cependant pour pouvoir séparer un noyau d'ordonnancement de l'application et de l'architecture il est nécessaire d'employer un mécanisme d'interface.

Le module de construction de tâches (MCT) a été identifié dans le chapitre 4 comme le module responsable de la gestion d'un graphe de flot de données représentant l'exécution d'un programme parallèle. Dans le chapitre 5 nous avons identifié l'exécutif, le module responsable pour offrir une machine parallèle virtuelle sur laquelle le programme d'application est exécuté.

La figure 6.1 présente la structure d'un environnement doté d'un noyau d'ordonnancement et des interfaces définies dans les chapitres précédents. Dans ce chapitre nous présentons la structure de ce module implanté avec ces interfaces. Le noyau d'ordonnancement applicatif a été le sujet de deux articles : [27] et [31].

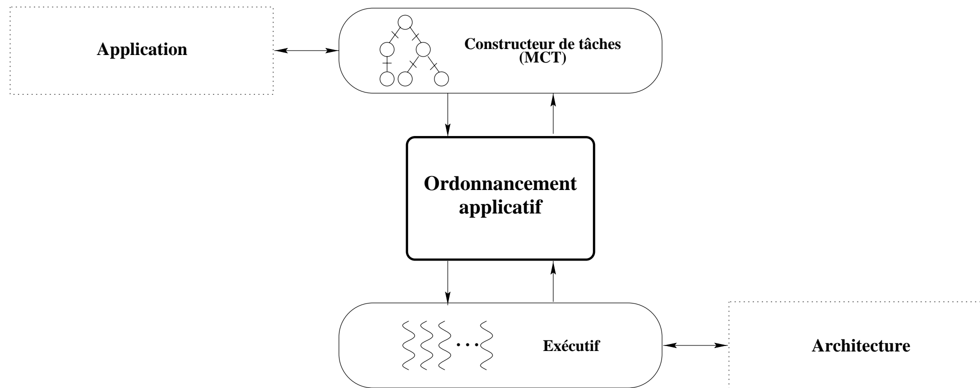

Figure 6.1 *Un environnement d'exécution parallèle doté d'un noyau d'ordonnancement générique. L'ordonnancement applicatif est composé de deux sous-systèmes : le sous-système de décision et le sous-système d'inspection de charge. La généricité du noyau est obtenue par la définition d'interfaces entre les niveaux application et architecture.*

Dans la section 6.2 nous présentons l'implantation des interfaces du noyau sur une architecture parallèle. Le problème de l'implantation du noyau d'ordonnancement composé d'un élément de décision et d'un élément d'évaluation de charge est considéré dans la section 6.3. La section 6.4 présente les classes qui composent la structure logicielle dans laquelle nous avons modelé le noyau d'ordonnancement générique. Finalement, dans la section 6.5, cette structure est validée par la modélisation des algorithmes de régulation de charge employés par les environnements étudiés dans le Chapitre 3.

6.2 L'interface du niveau d'ordonnancement

Nous présentons dans cette section les caractéristiques de l'implantation d'un noyau d'ordonnancement applicatif générique sur une machine parallèle réelle. La problématique d'implantation sur des architectures SMP et sur des architectures distribuées est associée, respectivement, à l'ordonnancement local et à l'ordonnancement global.

6.2.1 L'ordonnancement local sur une architecture SMP

Une architecture du type SMP est constituée d'un ensemble de processeurs dotés d'une mémoire commune. Tous les processeurs sont considérés comme des res-

sources de calcul identiques dédiées à l'exécution des tâches présentes dans le MCT. Or une telle architecture traduit parfaitement la notion de *pool d'exécution* (section 5.3.2). Par conséquent, un ordonnancement local est satisfaisant ici, puisqu'il n'est pas nécessaire de considérer la distribution de travail parmi les différents nœuds de l'architecture.

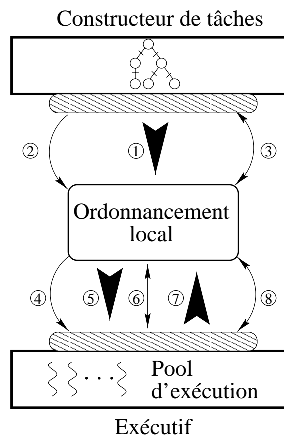

Figure 6.2 *Interface du noyau d'ordonnancement dans une architecture SMP. L'exécution des services est réalisée sur le flot d'exécution de l'appelant, sans que le noyau d'ordonnancement ait besoin d'un flot d'exécution propre.*

Dans ce cas, des processeurs virtuels sont créés par l'ordonnancement local. Ensuite l'ordonnancement donne la séquence d'exécution des tâches produites par l'application dans ce pool d'exécution. Les interactions entre l'ordonnancement, l'exécutif et le MCT, c'est-à-dire les demandes de services, sont représentées dans la figure 6.2. Un point important à observer concerne l'appel à n'importe quel des services présenté dans la figure : le traitement est réalisé de façon non bloquant. Cette règle permet d'éviter de situations de *deadlock*.

① Création de tâche. Le parallélisme au niveau de l'application est décrit par la construction de tâches. Par ce signal, le noyau d'ordonnancement prend connaissance du parallélisme qui a été généré. L'application peut fournir les tâches une à une, selon leur création, ou envoyer un ensemble de tâches.

② Changement d'état. Un signal est envoyé au noyau d'ordonnancement pour informer chaque altération dans le graphe dû à un changement d'état d'un nœud.

③ Parcours du graphe de flot de données. Les informations sur les relations de dépendances représentées dans le graphe peuvent être exploitées par la politique d'ordonnancement.

④ Configuration du pool d'exécution. Création et/ou destruction de processeur virtuel.

⑤ Démarrage de tâche. Lancement sur un processeur inactif d'une tâche prête. Le démarrage d'une tâche non prête serait une transgression de la sémantique spécifiée par le graphe de l'application.

⑥ Préemption de tâche. Dans la hiérarchie d'ordonnancement employée, l'exécution de tâches est supportée par des processeurs virtuels ordonnancés par un mécanisme offert par l'exécutif. Or l'ordonnancement applicatif n'a pas les moyens pour interagir directement avec cet niveau d'ordonnancement pour préempter le flot d'exécution d'un processeur virtuel. Ainsi c'est la tâche, à un instant quelconque de son exécution, qui informe l'ordonnancement applicatif qu'elle peut être préemptée. C'est seulement à ce moment que l'ordonnancement applicatif peut bloquer une tâche.

⑦ Fin d'exécution de tâche. Le processeur virtuel informe que la tâche qu'il exécutait est terminée et que, à partir de cet instant, il est inactif.

⑧ Lecture des informations de charge. Le noyau d'ordonnancement accède les informations sur l'utilisation des ressources de l'architecture.

6.2.2 L'ordonnancement global sur une architecture distribuée

Dans une implantation sur une architecture distribuée, le problème de l'ordonnancement se présente aussi au niveau global, c'est-à-dire la distribution de tâches parmi les nœuds de la machine parallèle. Cette couche d'ordonnancement est présentée par la figure 6.3 : l'ordonnancement global est présente sur tous les nœuds de la machine abstraite, ce qui permet que l'interaction entre le MCT et l'ordonnancement soit réalisée toujours localement par des appels de procédure.

L'abstraction de l'ordonnancement sur une machine distribuée est garantie par la maintenance d'un flux d'interactions interne au noyau d'ordonnancement : ce flux est destiné à la communication entre le niveau d'ordonnancement local, supporté par chaque nœud de la machine, et l'ordonnancement global. Ci-dessous sont présentées les opérations réalisées par l'ordonnancement global.

– *Placement de tâches et de données.* Un premier problème résolu au niveau de l'ordonnancement global est celui de l'attribution d'un site pour abriter

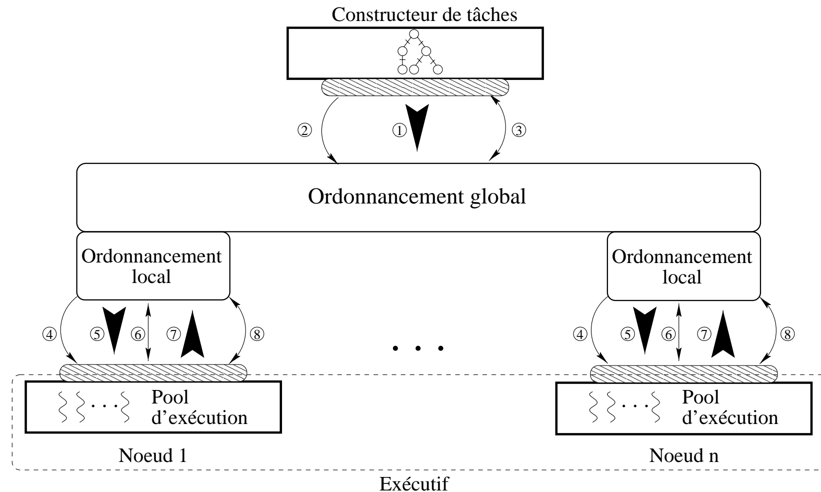

Figure 6.3 *Interface du noyau d'ordonnancement dans une architecture distribuée. La vision que l'application a du noyau d'ordonnancement dans une architecture distribuée est la même que sur une architecture à un seul nœud. Un niveau d'ordonnancement global prend en charge la distribution du travail parmi les nœuds de la machine.*

une donnée ou pour exécuter une tâche. Cette opération peut être réalisée à n'importe quel moment après la création de la tâche ou de la donnée.

- *Annulation de placement.* Après avoir placé une tâche ou une donnée, le niveau d'ordonnancement global conserve le droit de changer les sites des tâches et des données. Cette opération d'annulation de placement consiste à récupérer la tâche ou la donnée pour réaliser postérieurement un nouveau placement.
- *Migration de tâche en cours d'exécution.* Similairement à l'annulation de placement d'une tâche, l'ordonnancement global peut récupérer une tâche qui est en cours d'exécution et la replacer sur un autre site.
- *Lecture des informations de charge.* L'ordonnancement global a une connaissance de l'état global de la machine par l'accès aux informations de charge de chaque nœud.

Inexistence d'une mémoire globale. Contrairement à l'exécution sur une machine du type SMP, l'ordonnancement sur une architecture distribuée doit prendre en compte la localisation des données sur les modules de mémoire de chaque nœud. Même si le MCT est responsable de la gestion du graphe d'une façon distribuée [59], il n'est pas capable de décider la distribution des tâches et de don-

nées. L'ordonnancement doit donc prévoit le placement des données d'entrée d'une tâche τ sur le nœud où τ sera démarrée. Ainsi un nouvel état est observé pour les composantes du graphe : l'état *local*. Une donnée est considérée *locale* si elle est prête et, en plus, si ses données sont présentes sur le nœud en question ; une tâche est considérée *locale* si les données qu'elle prend en lecture sont locales au nœud dans lequel la tâche se situe. Donc, dans une architecture distribuée, la stratégie d'ordonnancement doit se préoccuper de placer les données nécessaires à une tâche sur le nœud où celle-ci sera exécutée : seules les tâches locales peuvent être démarrées. Ce nouvel état est ajouté aux états déjà définis dans le tableau 4.1 (section 4.5.2).

6.3 Implantation du noyau d'ordonnancement

Les actions réalisées par les deux niveaux d'ordonnancement, local et global, reflètent les décisions prises par une *politique d'ordonnancement*, cette politique d'ordonnancement est implantée sur deux sous-systèmes, un lié à la prise des décisions et un autre lié à l'inspection de la charge de la machine.

La structure interne du niveau d'ordonnancement sur un nœud et sa représentation sur une architecture distribuée est schématisé dans la figure 6.4. Dans une architecture distribuée, le niveau d'ordonnancement est présent sur chacun des nœud – la copie d'un module d'ordonnancement sur un nœud est nommée *réplicat*. Le principe des interactions prévoit des appels de procédures dans les échanges internes à un nœud, notamment entre le module de décision et le module d'inspection, et des envois de messages entre les réplicats.

Les modules de décision et d'inspection de charge sont présentés dans la suite de cette section ; nous abordons leurs responsabilités et les caractéristiques de leurs implantations pour l'ordonnancement d'applications parallèles. Une approche semblable à celle présentée ici peut être trouvée dans le travail de C. Jacqmot [73]. Dans la proposition de Jacqmot, des services pour les modules de décision et d'inspection sont identifiés et implantés ; ces services sont ensuite regroupés en « familles » : les associations entre une famille de décision et une famille d'inspection permettent la composition d'une politique d'ordonnancement. Dans notre approche seuls les services de l'ordonnancement pour les modules de décision et d'inspection sont identifiés, l'implantation de ces services est réalisée lors de l'implantation de la politique d'ordonnancement. Nous distinguons aussi les opérations réalisées au niveau d'ordonnancement global de celles réalisées au niveau local.

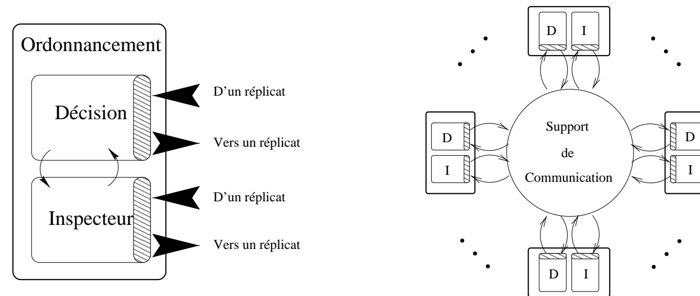

Figure 6.4 *Structure interne du niveau d'ordonnancement et les interactions entre les réplicats. Le noyau d'ordonnancement est composé d'un module de décision et d'un module d'inspection de charge. Sur chaque nœud de l'architecture est créé un réplicat d'un module d'ordonnancement. La communication entre les réplicats est possible grâce à des démons de communication employés par l'ordonnancement global.*

6.3.1 Le sous-système de décision

Le sous-système de décision est le responsable de l'implantation des politiques de déclenchement, de localisation et de transfert. Ce module est, entre autres, le responsable du contrôle de l'exécution des tâches sur la machine et du placement des données dans les modules de mémoire. Il est ainsi le point de contact de l'ordonnancement avec le niveau d'application, et le responsable de l'exploitation du graphe : chaque altération du graphe est informée par le niveau d'application sur le réplicat présent sur le nœud où l'altération a été produite.

Les opérations réalisées par ce module sont relatives aux actions d'ordonnancement local et global.

L'ordonnancement local. Au niveau de l'ordonnancement local, le module de décision considère la description locale du graphe et est responsable de la gestion du pool d'exécution et de la charge sur le nœud. La description locale du graphe comprend les tâches, les arêtes et les données présentes sur le nœud. Un élément du graphe est présent sur le nœud s'il a été créé localement et n'a pas souffert une opération de placement sur un autre nœud ou s'il a été créé sur un autre nœud mais placé sur le site.

L'ordonnancement global. Au niveau de l'ordonnancement global il est possible de considérer le graphe complet et les charges enregistrées sur l'ensemble des nœuds de la machine. Le degré de connaissance du graphe et de l'état de la machine

manipulée par le module de décision dépend de la politique d'ordonnancement supportée. Une politique peut très bien se contenter de réaliser un ordonnancement global « aveugle » – c'est le cas de Cilk, où un nœud est choisi au hasard pour être la source de la procédure de migration de tâche – ou prenant en compte des critères de charge de la machine, comme en GTLB qui comptabilise le nombre de tâches en exécution par nœud.

En tout cas, pour n'importe quel algorithme implanté sur l'ordonnancement global, des interactions entre les réplicats sont nécessaires. Comme les besoins de communication sont spécifiques à chaque algorithme d'ordonnancement, nous définissons seulement un mécanisme général pour les échanges entre les réplicats. Il est considéré que la réalisation d'une opération de l'ordonnancement global a pour objectif le transfert d'une certaine quantité de travail d'un nœud à un autre. Pour pouvoir identifier les nœuds source et destination de ce transfert, il est nécessaire d'avoir coopération entre au moins deux réplicats du module de décision. Ainsi, à chaque réplicat du module de décision sont associés deux *démons* par lesquels il est possible de réaliser l'interaction entre les réplicats par l'échange des messages.

– *Démon d'envoi* : même si les opérations de l'ordonnancement global sont appliquées sur l'ensemble de la machine, c'est sur un des nœuds que le besoin de déclencher une opération d'ordonnancement est vérifié. Le module de décision de ce nœud envoie donc un message à ses réplicats distants pour les enrôler dans l'activité d'ordonnancement. Le message envoyé et le nombre de réplicats enrôlés dans une activité d'ordonnancement global dépend de la stratégie appliquée.
– *Démon de réception* : c'est l'homologue du démon d'envoi, responsable dans un réplicat de recevoir les messages et de réveiller le module de décision pour l'opération d'ordonnancement global en cours.

Chaque démon peut avoir deux états : actif ou inactif. Un démon est activé explicitement pour réaliser un service pour l'ordonnancement global. Un démon actif est associé à un processus léger, qui partage avec le pool d'exécution l'accès aux processeurs réels ; un démon sert seulement une réquisition de service à la fois.

6.3.2 Le sous-système d'inspection de charge

Le sous-système d'inspection de charge (ou de manipulation d'information) est le responsable de l'enregistrement de l'état d'utilisation des ressources de la machine. Plusieurs indices de charges peuvent être contrôles par ce sous-système. Cependant, pour une politique d'ordonnancement seul un sous-ensemble de ces indices peut se montrer représentatif. D'ailleurs, un autre ensemble d'informations

peut être maintenu dans le module d'inspection de charge : les informations concernant une « trace » des décisions prises par le sous-système de décision, comme par exemple le nombre de tâches prêtes à démarrer mais qui sont gardées en réserve, ou encore le nombre de tâches non prêtes.

L'inspection de charge est ainsi implanté de façon à répondre aux besoins de la politique d'ordonnancement. Nous définissons pour le niveau d'inspection de charge une interface permettant sa configuration de façon à manipuler les informations de charge en fonction de la politique d'ordonnancement. Cette interface est composée fondamentalement de quatre services :

- *Création d'indice* : pour indiquer une fonction d'évaluation d'un indice de charge.
- *Création de voisinage* : pour définir les nœuds avec lesquels les informations de charge sont partagées.
- *Fréquence d'échange* : pour définir la fréquence des échanges des informations de charge entre les nœuds du voisinage.
- *Lecture d'indice* : pour que le module de décision puisse accéder à un indice de charge local au nœud ou aux indices de charge des nœuds du voisinage.

À l'exemple du sous-système de décision, le sous-système d'inspection doit lui aussi être supporté sur une architecture distribuée par le mécanisme de réplicats. Le mécanisme de démons de réception et d'envoi est donc utilisé pour les échanges des indices de charge entre les nœuds du voisinage.

L'ordonnancement local. L'ordonnancement local prend en compte la collecte et l'annotation des informations concernant la charge du nœud. Pour ne pas surcharger le nœud, la collecte des indices est restreinte aux indices exploitables par la politique de décision.

L'ordonnancement global. Au niveau de l'ordonnancement global, il est possible d'avoir une connaissance des informations de charge d'un ensemble de nœuds, voire de tous les nœuds de la machine. Du point de vue d'un réplicat du module d'inspection, cet ensemble correspond à son voisinage, c'est-à-dire les réplicats avec lesquels il échange des informations de charge. Une fonction d'évaluation permet d'identifier si un nœud est sur ou sous-chargé par rapport son voisinage.

6.3.3 Contrôle centralisé *vs.* contrôle réparti

Dans la discussion sur l'implantation du niveau d'ordonnancement présentée dans les sections précédentes, il est supposé que tous les réplicats sont identiques.

Chaque réplicat peut agir indépendamment et dans le cas d'une opération d'ordonnancement global être réalisée, une interaction entre les réplicats est alors nécessaire.

Cependant, cette structure peut être légèrement altérée pour élire un réplicat qui sera considéré comme un réplicat *centralisateur*. Sur ce réplicat sont centralisées les actions des modules de décision et/ou d'inspection au niveau de l'ordonnancement global. Ainsi, il est garanti que sur un certain réplicat il est possible d'avoir une vision globale des activités de la machine parallèle.

D'une façon semblable il est possible de créer une structure hiérarchique pour le niveau d'ordonnancement. Dans ce cas, un nœud est responsable pour centraliser les opérations d'ordonnancement d'un groupe de nœuds, étant lui aussi soumis aux décisions d'ordonnancement d'un autre nœud.

6.4 Implantation par classes

Pour décrire l'environnement d'exécution parallèle, nous avons conçu une structure logicielle en *classes*[1] pour représenter les différents modules identifiés. Ainsi trois classes ont été définies pour la construction d'un environnement d'exécution parallèle :

- La classe `MCT` pour implanter l'interface entre l'application et le noyau d'ordonnancement. Cette classe offre un ensemble de services au niveau applicatif pour la gestion des tâches produites, par exemple par la manutention d'un graphe de tâches. Du côté noyau d'ordonnancement, les services offerts permettent l'exploitation des tâches par une politique d'ordonnancement.
- La classe `Ordonnancement` pour offrir support à l'opération des politiques d'ordonnancement.
- La classe `Exécutif` pour offrir au noyau d'ordonnancement l'ensemble des services nécessaires à la création d'une machine abstraite et au contrôle de l'exécution d'un programme sur cette machine.

Les méthodes offerts par ces classes correspondent aux services identifiés dans les chapitres 4 et 5 et présentés dans la section 6.2 (figure 6.2).

Dans cette structuration par classes se retrouve les critères de généricité recherchés pour le noyau d'ordonnancement. D'une part il est possible d'utiliser le noyau spécifié sur différents architectures et interfaces de programmation ; pour cela il est nécessaire que soit fournies les spécialisations pour les classes `MCT` et `Exécutif`.

[1]Conception orienté objet pour le terme *classe*.

D'autre part la classe `Ordonnancement` offre seulement les ressources nécessaires au support à l'exécution d'une politique d'ordonnancement. Cette politique est implanté par deux classes :

– La classe `Décision`, pour implanter le sous-système de décision.
– La classe `Inspecteur`, pour implanter le sous-système d'inspection de charge.

Ainsi lors d'une phase d'initialisation du noyau d'ordonnancement, l'objet de la classe `Ordonnancement` crée les objets `Décision` et `Inpection` qui doivent être employés pour la stratégie d'ordonnancement.

6.5 Validation de l'environnement générique

Pour valider le noyau d'ordonnancement proposé nous présentons dans cette section une brève discussion sur l'implantation des stratégies d'ordonnancement des environnement présentés dans le chapitre 3. Nous détaillons l'implantation de quelques unes de ces stratégies dans le chapitre 8.

Équilibrage de charge de tâches indépendantes (GTLB). Pour implanter cette politique le seul service requis au `MCT` est de signaler au noyau d'ordonnancement les créations de tâches réalisées au niveau d'application. L'ordonnancement maintient, au niveau global, le contrôle de la charge des nœuds et implante un mécanisme de migration (restreinte ou de tâches en exécution) pour réguler la charge entre les nœuds.

Owner compute rule (DPC++). Pour que cette politique puisse être implantée, il est nécessaire que le `MCT` puisse informer pour chaque tâche la donnée manipulée ; l'ordonnancement global, réalisé de façon centralisé, peut concentrer les informations de charge de la machine et réaliser le placement de tâches de façon à pouvoir exploiter la localité et contrôler la distribution de charge. Dans le cas de DPC++ aucun contrôle sémantique de l'exécution n'est offert, mais il est garanti qu'une donnée n'est accédée que par une tâche à la fois : l'ordonnancement local peut réaliser ce contrôle s'il vérifie que deux tâches accédant une même donnée ne seront pas exécutées au même temps.

Algorithmes gloutons (Jade, Cilk et Cid). Pour implanter les algorithmes gloutons il est d'abord nécessaire que le noyau d'ordonnancement maintienne une liste de tâches prêtes en attente de façon à ce qu'elles puissent être migrées entre

les nœuds pour réguler la charge. Dans le cas où l'algorithme glouton implante une stratégie plus élaborée pour le choix de la tâche à être migrée (par exemple la localité des données dans Jade ou la stratégie profondeur d'abord dans Cilk), le `MCT` doit être capable de fournir les informations nécessaires à la stratégie ; il doit, par exemple, représenter les tâches de l'application dans un graphe de tâches et permettre au noyau d'ordonnancement de l'exploiter pour obtenir la localité des données ou pour identifier l'ordre d'exécution des tâches.

Algorithmes statiques (DSC, algorithme de regroupement). Pour les algorithmes d'ordonnancement statiques appliqués sur des graphes décrivant les relations entre les tâches, le `MCT` doit être en mesure de créer un tel graphe à partir des programmes exécutés au niveau applicatif. Pour l'exécution, l'ordonnancement local doit soit démarrer les tâches dans l'ordre calculé (DSC), soit démarrer toutes les tâches pour éviter toute situation d'interblocage entre les tâches communicantes (algorithmes de regroupement, Metis/SCOTCH).

6.6 Conclusion

Nous avons présenté dans ce chapitre un noyau générique pour l'ordonnancement. Les activités liées à chaque sous-système d'une politique d'ordonnancement, le module de décision et le module d'inspection de charge, ont été identifiées selon leur champs d'action – niveau global ou local – et leurs interactions. Nous avons pu constater qu'une structure commune à l'ordonnancement existe, bien que les actions associés à chaque élément de cette structure sont dépendantes de la solution recherchée.

Ce noyau générique a été validé par l'identification de la structure d'ordonnancement proposée ici sur des outils fournissant leurs propres mécanismes d'ordonnancement. Différentes politiques d'ordonnancement ont été ainsi considérées.

En utilisant la structure de ce noyau générique et les interfaces définies pour le niveau d'ordonnancement (chapitres 4 et 5), nous avons créé une structure logicielle offrant les conditions nécessaires pour implanter un environnement générique d'ordonnancement pour un langage de programmation parallèle.

Nous finissons ainsi la deuxième partie de ce document, dédiée à la description d'un environnement logiciel pour l'ordonnancement de programmes parallèles. Dans la partie suivante nous allons mettre en œuvre cet environnement.

Troisième partie

Ordonnancement pour ATHAPASCAN

Avant propos

Dans la deuxième partie de ce document nous avons caractérisé un environnement d'exécution de programmes parallèles doté des ressources d'ordonnancement. La mise en œuvre d'un tel environnement est présentée dans cette troisième partie.

Nous commençons par le chapitre 7, dédié à l'implantation d'un environnement d'ordonnancement en présentant le contexte dans lequel nous travaillons : ATHAPASCAN, un environnement de programmation parallèle à destination d'applications irrégulières. Ce chapitre illustre donc de manière pratique les discussions menées dans les chapitres 5 et 4 sur les interfaces entre l'ordonnancement et l'architecture, l'exécutif, et entre l'ordonnancement et l'application, le MCT.

Dans le chapitre 8 nous présentons l'implantation d'un environnement d'ordonnancement au sein de l'environnement ATHAPASCAN. Nous retrouvons ici les caractéristiques d'un environnement général présentées dans le chapitre 6.

Finalement, dans le chapitre 9, nous présentons une évaluation de l'environnement construit. Nous présentons quelques applications parallèles exécutées sur différentes politiques d'ordonnancement. Nous présentons aussi une visualisation de l'exécution d'une politique d'ordonnancement.

7

Integration de l'ordonnancement au sein d'ATHAPASCAN

Nous présentons dans ce chapitre les outils que nous avons utilisés dans l'implantation du module d'ordonnancement générique défini dans la deuxième partie de ce travail. Deux outils sont considérés : ATHAPASCAN-1 pour la programmation parallèle, permettant la construction dynamique d'un graphe de flots de données (cf. le chapitre 4), et ATHAPASCAN-0 un noyau exécutif offrant les fonctionnalités d'une architecture parallèle (cf. le chapitre 5). Pour ces deux outils, nous identifions les caractéristiques de leurs implantations et les échanges de services réalisées entre eux et le niveau d'ordonnancement.

Nous renvoyons le lecteur à la figure 1.2 (page 17), dans le chapitre d'introduction, où l'environnement de programmation ATHAPASCAN a été présenté. Nous centrons nos efforts dans la définition des interfaces du niveau d'ordonnancement avec ATHAPASCAN-0 et avec ATHAPASCAN-1. La discussion sur l'implantation du noyau d'ordonnancement est présentée dans le chapitre 8.

L'interface de programmation ATHAPASCAN-1 est présentée dans la section suivante. Ensuite, dans la section 7.2, nous présentons le noyau exécutif fourni par ATHAPASCAN-0. Tant pour ATHAPASCAN-1 que pour ATHAPASCAN-0 nous examinons les caractéristiques en prenant en compte les nécessités de l'ordonnancement.

7.1 ATHAPASCAN-1 et l'ordonnancement

Cette section est inspirée de l'article [58].

ATHAPASCAN-1 [58, 43, 59] est un langage de programmation parallèle permettant la construction distribuée dynamique d'un graphe de flot de données associé à l'exécution d'un programme. L'implantation d'ATHAPASCAN-1 est une bibliothèque C++, néanmoins une extension de C est manipulée par un précompilateur, permettant un codage plus simple. Le parallélisme en ATHAPASCAN-1 est exprimé par des appels à distance asynchrones de procédures, dénommées *tâches*, qui communiquent et sont synchronisées seulement par l'intermédiaire de l'accès à une mémoire partagée.

7.1.1 ATHAPASCAN-1 : l'interface applicative d'ATHAPASCAN

Un programme ATHAPASCAN-1 est composé d'un ensemble de tâches qui accédent des données dans un espace mémoire partagée. La sémantique de l'exécution est basée sur l'accès aux données partagées : la valeur obtenue par un accès en lecture d'une donnée partagée correspond à la valeur de la dernière écriture (selon l'ordre séquentiel) de cette donnée. Cette sémantique naturelle définie comme « lexicographique » dans [58], permet de sérialiser les opérations de lecture et d'écriture sur une donnée partagée. Cette sémantique naturelle est illustrée sur un exemple de programme en ATHAPASCAN-1 est présenté dans la figure 7.1.

```
task update( shared( w )< int > x )
     { x.write( 5 ); }
task print( shared( r )< int > x )
     { printf( "%d", x.read() ); }

task test() {
  shared< int > a;
  fork update( a );
  fork print( a );
}
```

Figure 7.1 *Un exemple de programme* ATHAPASCAN-1. *La sémantique d'exécution est basée sur l'ordre lexicographique d'accès aux données : dans cet exemple, la valeur* `5` *est obtenue en sortie par l'exécution de la tâche* `test`. *L'exécution de la tâche* `print(a)` *est retardée jusqu'à que la tâche* `update(a)` *soit exécutée. L'instruction* `fork` *crée des tâches de façon asynchrone et* `shared` *crée une donnée dans la mémoire partagée ;* `r` *et* `w` *informent, respectivement, que la tâche prend la donnée pour la lire ou pour l'écrire.*

Le contrôle de la sémantique d'accès est réalisé lors de l'exécution, c'est-à-

dire qu'un graphe de flot de données représentant les contraintes entre les tâches et les données est créé dynamiquement, au fur et à mesure que l'exécution avance. L'environnement d'exécution ATHAPASCAN-1 obtient les informations nécessaires pour maintenir ce graphe à partir de la définition d'une tâche, où sont spécifiées les données en lecture et en écriture de cette tâche. Une donnée dans la mémoire partagée peut être écrite ou lue par une tâche selon la spécification `w` (*write*) ou `r` (*read*), respectivement. Ainsi à chaque création de tâche, un nouveau nœud tâche est introduit dans le graphe : ses arrêtes d'entrée spécifient ses données en lecture et les arrêtes de sortie ses données en écriture. Les versions de données sont inserées dans le graphe d'une façon équivalente.

Les bases d'ATHAPASCAN-1 sont définies dans [104, 103]. Nous retrouvons dans le travail de M. Doreille [43] la spécification du langage ATHAPASCAN-1 et dans celui de F. Galilée [59] la description de son implantation sur une architecture distribuée. Une vue d'ensemble des travaux sur ATHAPASCAN-1 est donnée dans l'article [58].

7.1.2 L'interface avec l'ordonnancement

L'exécution d'un programme ATHAPASCAN-1 peut être suivie par l'évolution de son graphe de flot de données. Le noyau d'ordonnancement profite des informations contenues dans ce graphe pour implanter une politique d'ordonnancement. Ainsi, ATHAPASCAN-1 met à disposition de l'ordonnancement (cf. figure 7.2) un ensemble de services d'exploitation du graphe[1] :

- `RetourneTâcheLectrice` et `RetourneTâcheÉcrivain` : ces deux services permettent d'obtenir la liste des tâches qui lisent ou écrivent dans une donnée.
- `RetourneDonnéeLecture` et `RetourneDonnéeÉcriture` : par ces deux services le noyau d'ordonnancement peut d'obtenir la liste de données qui sont lues ou écrites par une tâche.
- `RetourneÉtat` : retourne l'état des tâche et des données face à ces prédécesseurs.
- `RetourneValeurs` : retourne les informations (définies par le programmeur) concernant les coût des tâches et des données.

Toute modification du graphe (ajout destruction ou modification de l'état d'un nœud) peut entraîner une décision d'ordonnancement. Ainsi, chaque modification doit être rapportée au niveau d'ordonnancement. ATHAPASCAN-1 dispose de trois services pour actionner l'ordonnancement :

[1] Cet ensemble de services est représenté dans la figure 7.2 par `LectureGraphe`.

– `NouvelleTâche` et `NouvelleDonnée` : pour informer qu'une nouvelle tâche, ou donnée, a été aditionnée au graphe.
– `NouvelÉtat` : pour informer qu'une tâche ou une donnée a changé d'état.

Finalement un dernier ensemble de services permet au noyau d'ordonnancement de demander à ATHAPASCAN-1 le déplacement de tâches et de données entre les nœuds d'une architecture distribuée :

– `DéplacerTâche` et `DéplacerDonnée` : permettent au noyau d'ordonnancement de soliciter à ATHAPASCAN-1 le transfert d'une tâche ou d'une donnée d'un nœud à l'autre dans une architecture distribuée.

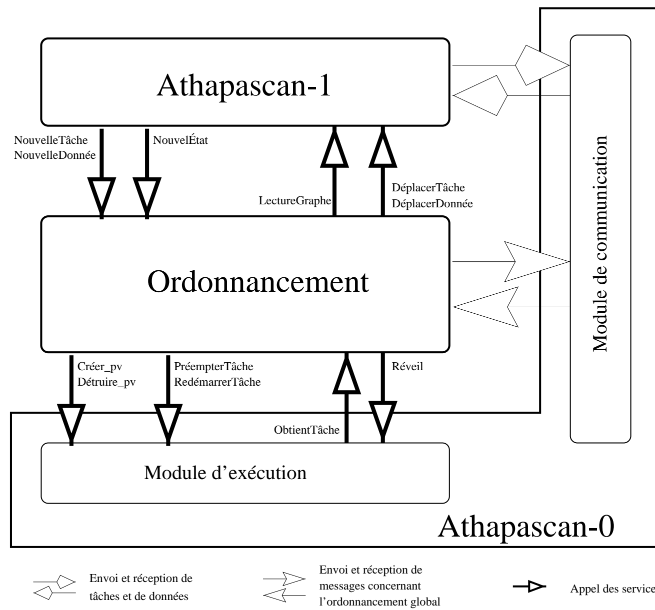

Figure 7.2 *Services requis et fournis par le niveau d'ordonnancement. Dans l'environnement d'exécution parallèle* ATHAPASCAN, ATHAPASCAN-1 *est une interface applicative qui dispose d'un module permettant la manutention d'un graphe de flot de données ;* ATHAPASCAN-0 *est un noyau exécutif offrant services de manipulation de processus légers et de communication.*

7.1.3 L'implantation

Les échanges entre le niveau d'ordonnancement et ATHAPASCAN-1 sont réalisées en fonction du graphe de flot de données : ATHAPASCAN-1 maintient la cohérence de ces informations pendant que le niveau d'ordonnancement l'exploite pour exécuter le programme selon une politique d'ordonnancement. Il est à noter que, dans une architecture à mémoire distribuée, ATHAPASCAN-1 maintient les informations concernant le graphe de manière distribuée parmi les nœuds. De cette façon, les échanges de services entre le niveau d'ordonnancement et ATHAPASCAN-1 sur un nœud ne concernent que la partie « locale » du graphe. Nous présentons dans la suite les composantes de ce graphe.

Le graphe est contitué par des « objets » contenant les données et les tâches qui concernent l'application. En effet, une tâche est représentée par une *clôture* et une donnée par une série de *versions*. Dans la figure 7.3 nous représentons un instant dans l'évolution du graphe de flot de données produit par l'exécution du programme dans la figure 7.1.

La clôture. Une clôture est la représentation d'une tâche dans le graphe. Elle contient le code devant être exécuté et les pointeurs vers les données qu'elle accède en lecture et les données qu'elle produit en sortie. Une information d'état est associée à chaque clôture. Cet état est calculé en fonction des données que la clôture prend en lecture et les opérations d'ordonnancement qu'elle a subies. Ainsi, une clôture peut être dans un des états suivants :

- `non prête` : la clôture attend ses données d'entrée qui ne sont pas encore disponibles.
- `prête` : la clôture dispose de ses données d'entrée et attend d'être démarrée.
- `locale` : la clôture est `prête`, mais en plus les données en lecture sont disponibles sur la mémoire locale du nœud.
- `en exécution` : la clôture est en train d'être exécutée.
- `terminée` : l'exécution de la clôture est terminée et ses données en sortie sont disponibles.

Les versions. Contrairement aux tâches, les valeurs des données sont altérées pendant l'exécution du programme. Ainsi, nous associons la notion de version pour représenter l'évolution d'une donnée pendant l'exécution du programme. Dans la figure 7.3 nous pouvons observer le mécanisme de versions : les clôtures `update`

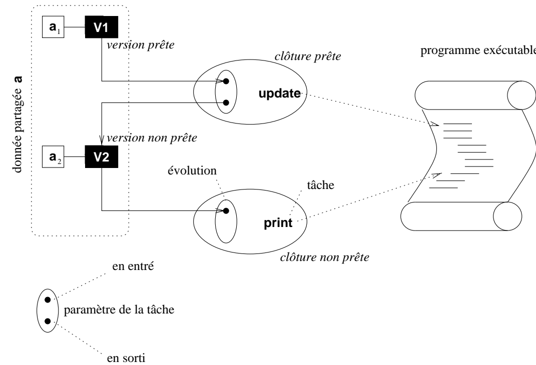

Figure 7.3 *Un instant dans l'évolution du graphe de flot de données. Une partie du graphe de flot de données correspondant à un moment de l'exécution du programme de la figure 7.1. L'instant représenté est celui juste après la création des tâches `update` et `print`, avant le démarrage de `update`.*

et `print` partagent l'accès à la donnée `a`, cependant, la clôtures `print` obtient en entrée la version `V2` de la donnée, dans ce cas, la valeur écrite par `update` en `a`. Ainsi les états exploités dans le graphe correspondent aux états des versions :

– `non prête` : la version ne possède pas encore de donnée associée.
– `prête` : une valeur a été associée à la version.
– `locale` : la version est `prête` et en plus, la valeur associée est présente sur le nœud.

7.1.4 Utilisation de l'ordonnancement applicatif

Nous présentons dans cet cette section la syntaxe pour l'utilisation du noyau d'ordonnancement à l'intérieur d'un programme ATHAPASCAN-1 ; dans l'annexe B nous présentons le code complet d'un programme ATHAPASCAN-1. Plus d'informations sur la syntaxe d'ATHAPASCAN-1 sont présentées dans [43].

Lorsque un programme ATHAPASCAN-1 est démarré, le noyau d'ordonnancement est initialisé avec une stratégie défaut[2] ; d'autres stratégies sont à disposition du programmeur dans une bibliothèque de *politiques d'ordonnancement*. Ainsi, de nouvelles stratégies peuvent être ajoutées explicitement dans le noyau d'ordonnancement par la création d'un objet correspondant à une classe qui définit une politique de décision souhaitée. Cette création suit la syntaxe C++ :

[2]Dans la version actuelle un algorithme de liste (section 8.3.2).

```
politique grp;
```

Par cette ligne de code, un objet de la politique d'ordonnancement `politique` est créé dans le noyau d'ordonnancement et son accès est réalisée par l'objet `grp`.

Par défaut en ATHAPASCAN-1, toute tâche créée est ordonnancée par la même stratégie qui a été utilisée pour ordonnancer la tâche qui l'a créée. La sélection d'une autre stratégie est réalisée par annotation dans le code. Ceci implique spécifier la politique à utiliser pour ordonnancer une tâche au moment de sa création (`Fork`). Cette spécification peut être faite de deux manières[3] :

1. Par le changement de la politique par défaut. Un appel à :
   ```
   a1_set_scheduling( [grp] );
   ```
   définit que la stratégie associée à `grp` doit être utilisée comme la stratégie défaut lors des prochains appels à `Fork` dans la tâche courante.
2. Par annotation d'un appel à l'opération `Fork` :
   ```
   Fork<[tâche]>( [grp] )([paramètres]);
   ```
   la stratégie associée à `grp` sera utilisée pour ordonnancer la tâche `tâche`.

De plus, les tâches peuvent fournir des informations additionnelles exploitables par les stratégies d'ordonnancement. Ces informations sont fournies aussi par annotation :

```
Fork<[tâche]>( [grp], SchedAttributs([info]) )([paramètre]);
Fork<[tâche]>( SchedAttributs([info]) )([paramètres]);
```

Dans ce cas, `info` contient des informations décrivant les caractéristiques propres à l'exécution de la tâche `tâche`. Ces informations sont ajoutées par ATHAPASCAN-1 dans le graphe de flot de données et pourront être exploitées par la politique d'ordonnancement. Exemples d'`info` : le coût d'exécution de la tâche ou sa priorité.

7.2 ATHAPASCAN-0 et l'ordonnancement

ATHAPASCAN-0 [20, 95, 22] est un noyau exécutif pour les machines parallèles supportant la programmation par des processus légers[4] communicants. L'utilisation d'ATHAPASCAN-0 se fait par l'intermédiaire d'une bibliothèque de fonctions implantée en C.

La portabilité a été un des aspects recherchés lors de l'implantation d'ATHAPASCAN-0. Il a été ainsi implanté sur des outils standards pour la manipulation des

[3] Dans notre notation, les caractères < et > sont obligatoires et les caractères [et] indiquent une information qui doit être fournie.

[4] Dans ce contexte, un processus léger est un *thread*.

processus légers et des communications. Les versions d'ATHAPASCAN-0 utilisées dans ce travail ont été implantées sur des bibliothèques de processus légers conformes à la norme POSIX [1] et sur des bibliothèques de communication conformes à la norme MPI [110].

7.2.1 ATHAPASCAN-0 : le noyau exécutif d'ATHAPASCAN

L'abstraction de la machine parallèle offerte par ATHAPASCAN-0 est une architecture composée de nœuds de calcul reliés par un réseau de communication. Un *nœud* ATHAPASCAN-0, dans les versions considérées, est constitué par un processus lourd Unix. La création de la machine ATHAPASCAN-0 est réalisée par le démarrage sur chaque nœud de la machine réelle[5] d'un nœud ATHAPASCAN-0.

Le parallélisme dans cette architecture est exploité à deux niveaux : interne au nœud et entre les nœuds.

Parallélisme intra-nœud. Le parallélisme intra-nœud est exploité par ATHAPASCAN-0 par la présence dans un nœud d'un ensemble de processus légers actifs. Ces processus légers s'exécutent en concurrence : un mécanisme d'ordonnancement garantit l'accès des processus légers aux processeurs réels du nœud. Un ensemble de primitives de synchronisation, notamment verrous et sémaphores, sont disponibles pour synchroniser l'exécution des processus légers. Ce type de primitive est souvent utilisé pour gérer l'accès à la mémoire commune des processus légers. Un autre point à souligner c'est que le nombre de processus légers dans un nœud n'est pas forcément constant, la création et la destruction de processus légers est totalement dynamique.

Parallélisme inter-nœuds. Le parallélisme inter-nœuds en ATHAPASCAN-0 est exploite par la présence en chaque nœud de la machine réelle d'un noyau d'exécution ATHAPASCAN-0. Nous avons ainsi un modèle d'exécution du type SPMD[6], le même code du programme est placé sur chaque nœud de la machine. Cependant chaque nœud progresse indépendamment des autres. Les communications entre les nœuds sont réalisés par échanges de messages. Un ensemble assez complet de primitives de communication synchrones et asynchrones est disponible. Contrairement à la possibilité de créer et de détruire dynamiquement les processus légers, la machine ATHAPASCAN-0 est créée lors du lancement d'un programme et reste immuable jusqu'à sa terminaison[7].

[5] Un nœud de la machine peut être un SMP ou un monoprocesseur.
[6] Unique programme multiples données, ou en anglais *Single Program Multiple Data*.
[7] Contrainte introduite par MPI-1.

Un démon de communications. ATHAPASCAN-0 implante un mécanisme basé sur l'utilisation d'un *démon de communication* sur chaque nœud pour faire avancer les communications. Ce démon (*i*) concentre les requêtes d'envois de messages du nœud et les envoie à leurs destinations, et (*ii*) scrute le réseau à la recherche des messages destinés au nœud et les met à disposition des activités réceptrices. Ce mécanisme a la particularité d'assurer que, lors de l'appel d'une primitive de communication bloquante, les ressources associées à son exécution ne restent pas bloquées, interdisant l'avancement des autres processus légers. La sérialisation des communications est réalisée par le démon[8]. Le problème est dans la définition de la fréquence avec laquelle le démon scrute le réseau, nous abordons ici le problème de la « réactivité » : étant implanté comme un processus léger, ce démon partage avec les activités du programme l'accès aux processeurs du nœud. Ainsi, moins les intervalles entre chaque scrutation du démon seront longs, plus le démon sera réactif pour traiter les échanges des messages, mais aussi plus de temps processeur est consommé par lui, au détriment des activités utilisateur.

7.2.2 L'interface avec l'ordonnancement

Dans l'interface entre le niveau d'ordonnancement et ATHAPASCAN-0 nous distinguons deux types de services : ceux destinés à la maintenance d'un pool d'exécution sur un nœud et ceux destinés aux échanges entre les nœuds (figure 7.2).

Le pool d'exécution. Un couple de primitives dans le noyau exécutif permet de définir le nombre maximal de flots d'exécution en parallèle :

– `Créer_pv` : pour ajouter un flot d'exécution.
– `Détruire_pv` : pour détruire un flot d'exécution.

Le nombre maximal d'activités concurrentes est défini par le nombre d'appels réalisés à `Créer_pv`, moins le nombre d'appels réalisés à `Détruire_pv`.

Le noyau exécutif cherche dans le niveau d'ordonnancement une à une des tâches à exécuter, dans la limite du nombre d'activités concurrentes spécifiées :

– `ObtientTâche` : est un service du niveau d'ordonnancement qui retourne une tâche utilisateur devant être exécutée.

Dans le cas où aucune tâche n'est retournée par cette fonction, le noyau exécutif bloque un flot d'exécution du pool d'exécution. Le « réveil » des flots d'exécution bloqués est réalisé explicitement par le niveau d'ordonnancement :

[8] Un tel mécanisme fourni un schéma de communication *thread-aware* [22].

– `Réveil` : remet en activité tous les flots d'exécution du pool.

Cet ordre de réveil indique qu'au moins une tâche utilisateur attend son exécution sur le nœud.

Les communications. Les services de communication sont employés par le niveau d'ordonnancement pour réaliser les opérations liées à l'ordonnancement global entre ses réplicats[9]. Pour les communications, les primitives utilisées sont :

- `EnvoiAsync` : pour envoyer un message à un destinataire spécifié. L'envoi est asynchrone, l'émetteur n'attend pas la réception du message par le destinataire[10].
- `RéceptionSync` : des services spécifiques sont associés aux traitements des messages reçus. Ces services sont activés lors de la réception d'un message et immédiatement traités.

Nous rappelons que les transferts des tâches et des données, c'est-à-dire la manutention distribuée du graphe de flot de données, sont des opérations gérées par ATHAPASCAN-1.

7.2.3 ATHAPASCAN-0 comme support à l'ordonnancement

On trouve dans ATHAPASCAN-0 les ressources nécessaires dans un niveau exécutif pour réaliser l'ordonnancement des programmes parallèles. Nous utilisons seulement un sous-ensemble des ressources disponibles en ATHAPASCAN-0, composé des primitives de communication et de manipulation de processus légers. Les processus légers sont utilisés pour construire un pool d'exécution, les primitives de communication permettent à l'ordonnancement d'envoyer entre les nœuds des messages contenant des informations propres à la politique d'ordonnancement, ainsi que les informations de charge.

7.2.3.1 Le pool d'exécution

Le pool d'exécution défini dans la section 5.3.2 est implanté comme un ensemble de processus légers chargés d'exécuter les tâches de l'application. Nous appelons chacun de ces processus légers *processeur virtuel*. Le nombre de processeurs

[9]La notion de « réplicat » a été introduite dans la section 6.3.
[10]En effet, grâce au démon de communications, l'émetteur n'attend même pas le départ du message du nœud.

virtuels dans un nœud définit le nombre maximal d'activités parallèles concurrentes à un certain moment sur un nœud de la machine. Ce nombre est un choix de la politique d'ordonnancement en fonction des caractéristiques de l'architecture.

L'algorithme 7.1 présente la séquence d'opérations réalisées par un processeur virtuel. Dans la ligne 2 de cet algorithme, nous identifions son interaction avec l'ordonnancement pour obtenir une tâche à exécuter. Si aucune tâche n'a été obtenue, le processeur virtuel entre dans un état d'inactivité en attendant que de nouvelles tâches prêtes soient présentes sur le nœud. Le « réveil » des processeurs virtuels est lié à la présence d'une tâche locale dans le niveau d'ordonnancement. Il est important de noter que si plus d'un processeur virtuel est inactif dans un pool au moment d'un réveil, seulement un seul parmi eux obtiendra la tâche à exécuter, les autres retourneront à l'état d'inactivité.

Algorithme 7.1 Séquence d'opérations réalisées par un processeur virtuel du pool d'exécution.

```
1: tant que ( le programme n'est pas terminé ) faire
2:     t = ordo→ObtiensTâche()
3:     si ( t ) alors
4:         Exécution( t )
5:     sinon {Il n'y a pas de tâches locales}
6:         AttendRéveil
```

Avant de poursuivre notre discussion, il est important de remarquer qu'une telle implantation d'un pool d'exécution n'est possible que grâce au modèle de tâche ATHAPASCAN-1. Un modèle de tâche permettant des synchronisations internes au flot d'exécution pourrait engendrer une situation d'interblocage des processeurs virtuels.

Lors de l'implantation du pool d'exécution nous avons considéré deux possibilités, (*i*) soit implanter un processus léger exécutant des tâches dans une boucle, (*ii*) soit créer un nouveau processus léger pour chaque nouvelle tâche démarrée. L'expérience que nous avons menée dans la phase initiale de notre projet [28] a montré que le coût de création et destruction dynamique des processus légers était élevé. Une solution plus économique – en fonction de temps – était d'employer des processus légers implantant l'algorithme 7.1. Cependant cette solution introduit d'autres surcoûts, notamment liés à la gestion des synchronisations entre les processus légers et aux changements de contexte.

Un mécanisme semblable est présent dans Filaments [86]. Dans Filaments, sur chaque nœud de la machine sont maintenus des *processus léger serviteurs*, responsables de l'exécution des activités du programme d'application. Le programme en cours d'exécution crée des *filaments*, c'est-à-dire des tâches à exécuter – ces fila-

ments sont considérés comme des processus *très* légers[11]. Les filaments créés sont maintenus dans un *pool de filaments* en attendant leur exécution par un processus léger serviteur. Nous retrouvons dans [86] une comparaison entre les surcoûts dûs aux changements de contexte de processus légers traditionnels et les changements de contexte de filaments. Cette comparaison montre que les changements de contexte de filaments, dans le pire cas, sont 75 fois plus performants.

Tableau 7.1 *Temps liées aux opérations sur les processus légers et aux manipulations de verrous [22] sur deux architectures SMP sur Solaris 2.6. La colonne p^v vide représente le temps de création, d'exécution et de destruction d'un processus léger sans corps.*

Architecture	p^v vide	changement de contexte	*lock\unlock*	fonction vide
2 × 333 Mhz Pentium II	160 μs	12.54 μs	0.347 μs	0.034 μs
4 × 200 Mhz Pentium Pro	210 μs	17.72 μs	0.57 μs	0.058 μs

Pour illustrer notre choix, nous analysons le coût d'exécution d'une tâche sur le pool. Le tableau 7.1 présente des résultats présentés par A. Carissimi dans [22][12] pour quelques opérations élémentaires dans la manipulation de processus légers. À titre comparatif, le temps d'appel d'une fonction vide (une fonction sans corps) est aussi présenté. Nous considérons qu'une tâche est disponible dans le noyau d'ordonnancement et que tous les q processeurs virtuels du pool d'exécution sont inactifs. Lors du « réveil » de ces q processeurs virtuels, le surcoût généré est de l'ordre de :

$$S_q = q \times (2 \times T_{cc} + T_{zc}) - T_{cc}$$

Où : T_{cc} est le temps d'un changement de contexte et T_{zc} le temps de sortie et entrée dans une zone critique (obtention de la tâche).

Ainsi, le surcoût du réveil d'un pool d'exécution sur le bi- et sur le quadri-processeur présentés dans le tableau 7.1 (avec respectivement $q = 2$ et de $q = 4$) est de l'ordre de $S_2 = 38.3\mu$s et $S_4 = 126.32\mu$s ; ces valeurs sont inférieures à celles de la création et destruction d'un processus léger.

L'ordonnancement du pool par ATHAPASCAN-0. ATHAPASCAN-0 n'introduit aucun modèle d'ordonnancement pour les processus légers. Ce niveau d'ordonnancement repose directement sur le support offert par la bibliothèque de processus

[11] Les filaments dans [86] sont comparés à des *very lightweight threads*.
[12] Solaris 2.6 et *threads* système.

légers utilisé, dans notre cas des bibliothèques POSIX. Le standard POSIX, ne défini pas une, mais trois politiques d'ordonnancement : une politique FIFO, une autre du type tourniquet et une troisième dépendante de l'implantation. Pour l'ordonnancement du pool d'exécution la politique d'ordonnancement utilisée à ce niveau est celle choisie par défaut par ATHAPASCAN-0, sans faire aucune hypothèse sur son fonctionnement – c'est-à-dire, la politique utilisée est une de celles disponibles dans l'implantation POSIX. Cependant nous savons que l'ordonnancement défaut sur Solaris 2.5 ou sur AIX 4.2, en utilisant les *threads* système, permet une distribution équitable du temps de processeur parmi les processus légers.

7.2.3.2 Les communications

Les communications emploient directement le démon ATHAPASCAN-0. Ainsi, le niveau d'ordonnancement a dans chaque nœud des « points de sortie » et des « points d'entrée » utilisés pour interagir avec le démon ATHAPASCAN-0 pour les communications entre les nœuds. Un couple composé d'un point de sortie et d'un point d'entrée dans des nœuds différents consiste en un canal d'activation de services de l'ordonnancement global.

Un envoi consiste à poster un message adressé à un nœud dans un point de sortie. Ce message est envoyé de façon asynchrone au point d'entrée correspondant au nœud destination. La réception implique le reveil d'un démon chargé d'exécuter une fonction de traitement associée au point d'entrée correspondant.

7.3 Conclusion

Dans ce chapitre nous avons présenté ATHAPASCAN-1 et ATHAPASCAN-0, respectivement les supports aux niveaux d'application et d'exécution de notre environnement de programmation parallèle. Nous avons identifié dans ces outils les services considérés nécessaires à un environnement de régulation de charge.

En ATHAPASCAN-1 nous avons retrouvé les caractéristiques identifiées comme nécessaires pour décrire un graphe de flot de données. Ce graphe peut être entièrement exploité par le noyau d'ordonnancement pour obtenir les informations de précédence et de localité sur les tâches et les données. ATHAPASCAN-1 fournit aussi des facilités de manipulation de tâches et données entre nœuds.

Avec ATHAPASCAN-0 nous avons à disposition toutes les fonctionnalités attendues au niveau de l'exécutif. D'ailleurs, ATHAPASCAN-0 nous offre une couche permettant le portage du noyau d'ordonnancement – et par conséquent d'ATHAPASCAN-1 – entre diverses architectures.

8

Mise en œuvre des stratégies d'ordonnancement

Dans ce chapitre est présentée la mise en œuvre de différentes stratégies d'ordonnancement sur le noyau d'ordonnancement applicatif que nous avons proposé pour un environnement d'exécution parallèle (cf. chapitre 6). Nous l'avons conçu de façon à avoir une interface générique pour l'implantation de diverses politiques d'ordonnancement.

Tout d'abord, dans la section 8.1, nous présentons comment nous avons modélisé le fonctionnement de l'ordonnancement applicatif en fonction des sous-systèmes de décision et d'inspection de charge. Ensuite, dans la section 8.2, nous présentons une discussion sur l'implantation des stratégies mixtes d'ordonnancement. Dans la section 8.3 est présentée l'utilisation de ce noyau dans l'environnement ATHAPASCAN pour l'implantation des politiques d'ordonnancement[1].

8.1 L'ordonnancement applicatif

Le noyau d'ordonnancement applicatif a été modélisé comme une « interface générique » de façon à pouvoir implanter différentes politiques d'ordonnancement. Cette interface générique consiste en un ensemble de fonctions correspondant aux interfaces du noyau d'ordonnancement présentées dans les chapitres 5 et 4. La sémantique associée à chacune de ces fonctions dépend entièrement de la politique

[1] Son utilisation dans un programme ATHAPASCAN-1 est présentée dans l'annexe B.

implantée. Ainsi, en pratique, l'implantation d'une politique d'ordonnancement correspond à l'implantation des opérations liées à chacune de ces fonctions.

Nous détaillons dans la suite les sous-systèmes de décision et d'inspection de charge.

8.1.1 Le sous-système de décision

Le sous-système de décision constitue le cœur du noyau d'ordonnancement. C'est lui qui explore et contrôle l'interprétation du graphe de l'application sur le pool d'exécution. Il est construit par une classe C++ composée de fonctions-crochet. Ces fonctions-crochet[2] représentent les services qui ont été prévus pour le noyau d'ordonnancement.

Dans la suite nous décrivons son fonctionnement dans l'environnement de programmation ATHAPASCAN-1.

8.1.1.1 Implantation d'une politique

La classe que nous avons définie pour le sous-système de décision est en réalité une classe virtuelle pure[3]. Cette classe virtuelle ne comporte aucune politique d'ordonnancement, c'est-à-dire que les corps des fonctions-crochet ne sont pas définis. L'implantation d'une politique consiste à fournir le code contenant les opérations d'ordonnancement qui doivent être réalisées lors de l'activation de chaque fonction-crochet. Ainsi, une nouvelle politique est définie par la spécialisation[4] de cette classe virtuelle par une classe qui fournit le code pour les fonctions-crochet.

Les fonctions-crochet à définir sont listées dans la suite. Les trois premières correspondent à l'interface avec le niveau applicatif et la dernière avec le pool d'exécution.

`bool NouvelleTâche( tch )` : cette fonction-crochet est appelée lors de la création d'une tâche. La tâche créée est identifiée par `tch`. Ce service permet aux algorithmes d'ordonnancement de réguler la granularité de l'exécution de l'application : si cette fonction retourne la valeur `vrai` cela signifie que le noyau d'ordonnancement a pris en charge l'exécution de la tâche. Si le service retourne `faux`, l'exécution de la tâche est laissée à la charge du niveau applicatif : la tâche est alors exécutée comme une fonction ordinaire sur le même flot d'exécution que la tâche

[2] En anglais : *hooks*.

[3] Nous présentons la définition C++ de cette classe dans l'annexe A.

[4] Nous employons pour le mot *spécialisation* le sens C++ lors de l'héritage de classes. L'expression *classe virtuelle* a d'ailleurs la même origine.

créatrice ; on parle alors de exécution séquentielle. Au début de de ce chapitre nous considérons que les tâches sont créées ; nous reviendrons sur la dégénérescence séquentielle dans la section 8.3.3.
Exemple : supposons une stratégie qui maintient des listes de tâches selon leur état. Ainsi trois listes sont manipulées : une liste de tâches en attente `L_attente`, une liste de tâches prêtes à l'exécution `L_prêt` et une liste de tâches locales `L_local`.

```
bool NouvelleTâche( tch )
si tch.état == attente alors
   L_attente.insère( tch )
si tch.état == prêt alors
   L_prêt.insère( tch )
si tch.état == local alors
   L_local.insère( tch )
retour vrai
```

`NouvelÉtat( tch )` : cette fonction-crochet est appelée lorsque l'état d'une tâche dans le graphe change. Ainsi, la politique d'ordonnancement peut savoir quand une tâche passe de l'état attente à l'état prêt et de l'état prêt à l'état local.
Exemple : considérons toujours une stratégie qui maintient des listes de tâches selon leur état.

```
NouvelÉtat( tch )
si tch.état == prêt alors
   L_attente.enlève( tch )
   L_prêt.insère( tch )
si tch.état == local alors
   L_prêt.enlève( tch )
   L_local.insère( tch )
```

`GrapheDécrit` : ce service est appelé pour signaler à la politique d'ordonnancement qu'une partie du graphe a été entièrement décrite. Ce service est spécialement utile lorsque la politique d'ordonnancement exploite un ensemble de tâches. C'est le cas de l'utilisation d'un algorithme DSC que nous présentons dans la section 8.3.4.
Exemple : une fonction `foo` prend en entrée listes de tâches selon leur états et retourne le placement de chaque tâche.

```
GrapheDécrit
foo( L_attente, L_prêt, L_local )
– appliquer placement –
```

`ObtientTâche` : ce service doit retourner une tâche qui est en mesure d'être exécutée immédiatement par le pool d'exécution, c'est-à-dire une tâche locale sur le nœud. Dans le cas où aucune tâche n'est locale, ce service retourne `nil`.
Exemple : pour exemplifier ce service supposons une stratégie qui exécute une tâche présente dans la liste de tâches locales ; si cette liste est vide, une demande de tâche est envoyée à un réplicat.

```
ObtientTâche
si L_local.taille > 0 alors
   tch → L_local.tête
   retour tch
sinon
   – envoit démande de tâche –
   retour nil
```

Nous avons implanté pour l'environnement ATHAPASCAN, un ensemble de politiques d'ordonnancement en fournissant un code aux fonctions-crochet décrites cidessus. Ces politiques sont disponibles sous la forme d'une bibliothèque de classes.

8.1.1.2 Initialisation du noyau d'ordonnancement

Le processus d'initialisation du noyau d'ordonnancement est réalisé entre le lancement du programme et l'exécution du premier calcul défini par le programmeur. Dans un premier temps est crée sur chaque nœud de la machine parallèle un réplicat du noyau d'ordonnancement. Ensuite un objet correspondant à la politique d'ordonnancement utilisée pour l'ordonnancement de tâches est créé à l'intérieur de chacun de ces réplicats[5]. Ainsi il est nécessaire de connaître la politique à utiliser avant l'exécution de la première instruction de l'application. Pour reconnaître la politique, nous avons utilisé le processus d'initialisation des objets statiques de C++.

L'utilisation de statiques en C++ permet l'exécution de code avant que le programme principal (la fonction `main`) ne soit commencé. Nous avons donc introduit une variable statique dans chaque classe de politique d'ordonnancement ; lors de l'initialisation de cette variable il est possible d'identifier la politique à introduire dans le noyau d'ordonnancement lors de son initialisation[6]. Ainsi la politique d'or-

[5]Nous verrons par la suite que plusieurs politiques d'ordonnancement peuvent être utilisées simultanément par un programme ATHAPASCAN-1.

[6]La norme C++ dit que tous les objets statiques référencés dans un programme sont créés ; nous nous basons sur cette création pour identifier la politique à utiliser. Cependant certains compilateurs créent toutes les variables statiques définies dans le programme (c'est le cas du compilateur XLC v. 3.4). Dans ce cas nous avons une situation où toutes politiques sont créées à l'intérieur du noyau d'ordonnancement, ce qui peut engendrer un surcoût additionnel dans la gestion du noyau.

donnancement à utiliser est définie dans le code du programme par la création d'un objet de la classe correspondant à la politique choisie[7].

8.1.1.3 Exécution des opérations d'ordonnancement

Les opérations d'ordonnancement sont réalisées lorsque le noyau d'ordonnancement est activé par un appel à un de ces services ou lorsqu'un message provenant d'un réplicat est reçu. Aucune opération n'est donc réalisée sans une activation externe. Cela nous permet d'implanter le noyau d'ordonnancement sans qu'un flot d'exécution propre à lui soit nécessaire.

Les fonctions-crochet sont exécutées comme des fonctions traditionnelles, en utilisant le flot d'exécution de l'appelant. En pratique nous observons que tous les appels aux fonctions-crochet sont supportés par les processeurs virtuels du pool d'exécution[8]. Il est à noter que l'envoi d'un message doit être réalisé de façon asynchrone : l'envoi synchrone d'un message peut provoquer une situation d'interblocage dans l'application.

De façon semblable un flot de données est réveillé par le démon de communication lors de la réception d'un message. Ce flot est responsable du traitement du message.

8.1.2 L'inspecteur de charge

Le sous-système d'inspection de charge a été modélisé comme un objet sur lequel est réalisé le contrôle des indices de charge de la machine. Nous avons conçu cet objet autonome pour réaliser les échanges d'indices de charge entre les réplicats (la politique de dissémination d'informations). Or comme les indices de charge manipulés lors de l'ordonnancement sont définis par la politique, l'objet d'inspection de charge doit être configuré pour répondre aux besoins de cette politique.

8.1.2.1 Configuration de l'inspection de charge

La configuration du sous-système d'inspection de charge est réalisée par la politique construite lors de l'initialisation du noyau d'ordonnancement. Dans cette phase de configuration sont créés les indices de charge qui seront utilisés par la

[7] Un mécanisme d'initialisation semblable, concernant l'identification globale de portes de communication, a été résolu dans ATHAPASCAN-0 par un mécanismes de nommage explicite.

[8] Dans la figure 9.4 (page 154) nous montrons le comportement du pool d'exécution ; nous pouvons observer que les opérations d'ordonnancement s'intercalent entre les opérations propres à l'exécution de l'application.

politique d'ordonnancement et le schéma des échanges d'informations entre les réplicats. La configuration du sous-système d'inspection de charge est réalisée par l'utilisation des trois services listés ci-dessous :

CréationIndice(ind_id) : un nouvel indice de charge, identifié par `ind_id`, est créé.

EnvoiIndice(foo, listevoisins) : définit le schéma pour les échanges des informations de charge entre les réplicats. La politique de déclenchement de la procédure d'échange d'information est spécifiée par la fonction `foo` ; les réplicats concernés par cette dissémination sont spécifiés par `listevoisins`.

RéceptionIndice(foo) : ce service permet de configurer une fonction qui devra être appelée lors de la réception d'un message. Cette fonction est généralement implantée dans l'objet politique d'ordonnancement. Une application typique de `foo` est d'implanter un mécanisme de régulation de charge entre les nœuds. Nous présentons dans la section 8.3.1 l'implantation d'une politique d'ordonnancement qui utilise une telle fonction.

8.1.2.2 Exécution des opérations de l'inspection de charge

L'interaction entre les sous-système de décision et le sous-système d'inspection de charge est réalisée par trois services :

AddV(ind_id, valeur) : pour additionner la valeur `valeur` à l'indice `ind_id`.

SousV(ind_id, valeur) : pour soustraire la valeur `valeur` à l'indice `ind_id`.

ObtientIndice(ind_id, réplicat) : ce service retourne la valeur de l'indice `ind_id` enregistrée localement pour le réplicat `réplicat`. Il est important de noter que cette valeur est plus ou moins à jour selon la fréquence des échanges d'indices de charge.

L'exécution des opérations du sous-système d'inspection de charge se fait d'une façon semblable à celui du sous-système de décision. Les opérations sont réalisées lors des appels aux services d'accès aux indices de charge ou de la réception d'un message contenant les informations de charge d'un autre réplicat.

Observation sur l'implantation. Nous n'avons pas implanté l'intégralité des fonctionnalités définies pour ce sous-système. En effet, les mécanismes de contrôle de charge pour les politiques que nous avons implantées sont assez simples, voir inexistants. Ainsi nous les avons intégrées à la politique de décision lorsque l'inspection de charge a été nécessaire.

8.2 Cohabitation des politiques d'ordonnancement

Dans cette section nous étudions la possible cohabitation de différentes politiques dans un noyau d'ordonnancement lors de l'exécution d'un programme. Nous présentons d'abord un cas où l'utilisation d'une « stratégie d'ordonnancement mixte » peut se montrer intéressante[9]. Ensuite, dans la section 8.2.2, nous présentons la solution que nous avons choisie pour implanter cette cohabitation.

8.2.1 Le problème

Nous présentons ici un cas dans lequel il peut se montrer intéressant d'utiliser une stratégie mixte d'ordonnancement. Nous basons notre discussion sur une application parallèle de type itération de Jacobi. Dans les derniers paragraphes de cette section nous proposons trois solutions pour implanter l'ordonnancement de cette application. Une étude semblable est présentée dans [94].

L'application. Supposons une application générant des tâches récursivement d'une façon telle que les tâches générées dans un niveau i soient dépendantes des données produites par le niveau $i-1$ (conforme au schéma de la figure 8.1)[10]. Nous pouvons observer que pour une telle structure de graphe, un bon ordonnancement peut être obtenu par la composition d'ensembles de tâches suivie de l'exécution de chaque ensemble par un nœud. De cette façon il est possible d'exploiter la localité des données en réduisant les communications entre les nœuds. De plus, cette stratégie permet une distribution équitable de la charge produite.

Or cette distribution peut se montrer inefficace si cet algorithme d'ordonnancement est appliqué sur une architecture dans laquelle la vitesse des nœuds est variable, puisque les nœuds plus lents peuvent retarder l'avancement de l'exécution. La solution pour retrouver l'efficacité peut impliquer l'implantation d'un algorithme supportant la migration de tâches : par exemple un algorithme glouton. Ainsi la quantité de tâches exécutées par un nœud correspond à la capacité de ces ressources. La localité des données n'est pas considérée, toutes les tâches sont considérées équivalentes dans la liste de tâches, et le nombre de communications entre

[9]Nous dénommons *stratégie mixte* une stratégie d'ordonnancement qui prend en compte plusieurs critères d'ordonnancement.

[10]Une évaluation de la performance de cette application est présentée dans la section 9.4 (page 144).

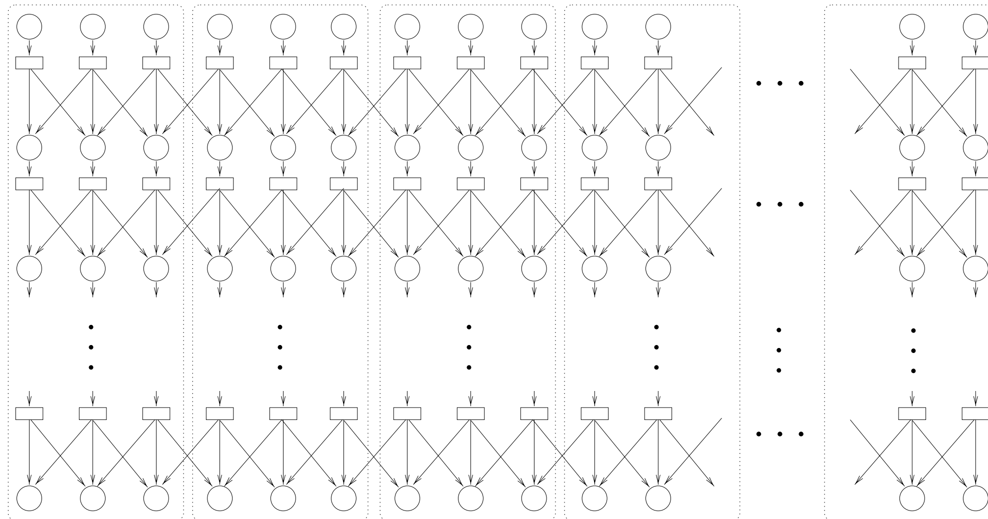

Figure 8.1 *Le graphe de flot de données d'une application parallèle de type Jacobi. Nous présentons ce graphe pour discuter le problème de l'implantation d'une stratégie d'ordonnancement mixte. Le programme lors de son exécution génère récursivement des tâches de même coût.*

les nœuds peut être considérable. Une idée est donc d'essayer de tirer bénéfice des deux politiques.

Nous abordons dans la suite les approches que nous avons trouvées pour implanter des stratégies mixtes.

Solution directe. La première solution que nous proposons à ce problème est l'implantation d'un unique algorithme d'ordonnancement qui prend en compte tant la localité des données que la distribution de tâches entre les nœuds. Ainsi, pour le problème de la figure 8.1, l'ordonnancement doit dans un premier temps regrouper les tâches de façon à distribuer des ensembles des tâches entre les nœuds et ensuite permettre que les tâches soient migrées[11]. Un problème qui se pose ici est d'identifier le moment où la politique d'ordonnancement doit changer de stratégie.

Solution par duplication. Une deuxième solution est d'implanter un mécanisme permettant à l'application de basculer entre deux noyaux d'ordonnancement,

[11]Il est à noter que lors de la migration de tâches, la localité de données peut être toujours exploitée.

chacun implantant une des politiques d'ordonnancement nécessaires au programme. C'est le niveau applicatif qui a la responsabilité d'activer l'un ou l'autre noyau selon la phase d'exécution. Dès que le noyau d'ordonnancement est changé, la nouvelle politique est appliquée à toutes les tâches, même celles déjà créées.

Solution par duplication partielle. Finalement, une troisième solution au problème considére un mécanisme interne au noyau d'ordonnancement permettant l'exécution simultanée de plusieurs politiques. Ce mécanisme implique une duplication des mécanismes implantant la politique d'ordonnancement, mais permet de garder un seul point de contact avec l'application. Dans ce cas chaque tâche soumise au noyau d'ordonnancement doit identifier la politique à être utilisée pour elle. Ainsi une tâche n'est associée qu'à une seule politique, dès sa création jusqu'à sa terminaison. Cette méthode est totalement compatible avec l'idée qu'une application est composée de phases de calcul et que chaque phase peut avoir des besoins propres en terme d'ordonnancement (cf. section 4.4).

8.2.2 Solution retenue

Pour implanter la cohabitation des politiques d'ordonnancement nous avons choisi la solution par duplication partielle. Ce choix permet le respect des besoins d'ordonnancement de chaque étape de l'application et aussi la définition de classes de politiques d'ordonnancement qui puissent être combinées pour implanter une stratégie mixte. Notons que ce choix laisse toujours possible la construction de politiques d'ordonnancement à stratégies multiples.

Le nouveau problème qui se présente est alors d'identifier la politique d'ordonnancement qui doit être appliquée à chaque tâche créée par le programme. Nous introduisons pour cela la notion de *groupe* de tâches. Un groupe est un objet qui est associé à une politique d'ordonnancement lors de sa création. Ainsi dès sa création une tâche est associée à un groupe, et toute interaction avec le noyau d'ordonnancement est réalisée via ce groupe.

En pratique le programmeur crée dans son code des groupes d'ordonnancement, chaque groupe associé à une politique d'ordonnancement (plusieurs groupes peuvent être associés à une même politique). Ensuite toutes les tâches sont associées à des groupes lors de leur création ; ceci permet au noyau d'ordonnancement, en identifiant le groupe auquel la tâche appartient, de la diriger vers la politique correcte.

Nous présentons cette nouvelle structure du noyau d'ordonnancement dans la figure 8.2. Notons que le sous-système d'inspection de charge n'a pas été répliqué pour permettre une vision cohérente de l'état de la machine par toutes les politiques d'ordonnancement.

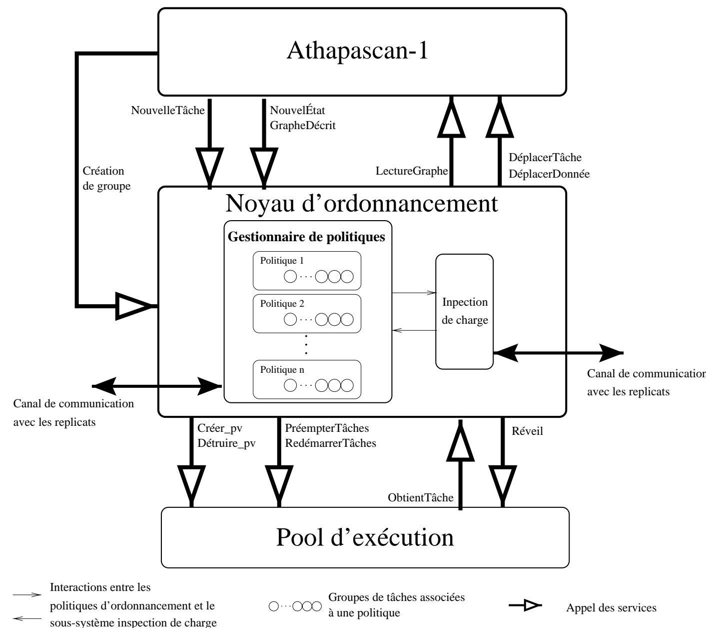

Figure 8.2 *Gestion de plusieurs politiques d'ordonnancement. Un gestionnaire de politiques prend en charge le contrôle des politiques d'ordonnancement dans le sous-système de décision. Un mécanisme de* groupes d'ordonnancement *est employé pour permettre d'associer les tâches générées par le programme aux politiques supportées par le noyau d'ordonnancement.*

Le gestionnaire de politiques de décision. Une fois que plusieurs politiques sont supportées par le noyau d'ordonnancement il est nécessaire de définir le mécanisme de partage du pool d'exécution. Nous avons adopté une règle simple : à chaque politique d'ordonnancement est associée une priorité pour l'exécution de ses tâches et le principe du premier arrivé, premier servi est appliqué lorsque deux politiques ont la même priorité. En pratique, lorsque le service `ObtientTâche` est appelé par le pool, la tâche choisie appartient à la politique la plus prioritaire.

8.3 Utilisation pratique

Comme nous avons vu précédemment dans ce chapitre (section 8.1.1), une politique d'ordonnancement est implantée par la spécialisation d'une classe virtuelle qui

définit les fonctions-crochet appelées lorsque les opérations d'ordonnancement sont nécessaires. Nous présentons dans cette section une utilisation pratique de cette interface par l'implantation de quatre politiques d'ordonnancement, trois dynamiques et une statique.

Nous présentons d'abord les algorithme dynamiques : dans la section 8.3.1 nous présentons un algorithme d'équilibrage de charge basé sur la migration de tâches en cours d'exécution (section 8.3.1) ; ensuite nous présentons deux algorithmes de partage de charge : un glouton (section 8.3.2) et un sur le vol de travail (section 8.3.3). Pour terminer, nous présentons une implantation d'une politique supportant l'algorithme de DSC (section 8.3.4).

Avant de rentrer dans la présentation des algorithmes, nous rappelons que nous différencions une tâche prête d'une tâche locale (cf. section 6.2.2, page 96). Une tâche locale est une tâche prête mais qui en plus a ses données sur le nœud. Seules les tâches locales peuvent être démarrées.

Le lecteur peut se rapporter à la figure 8.2 pour suivre l'emploi des services de l'interface.

8.3.1 Un algorithme d'équilibrage de charge

Nous considérons ici l'implantation d'un algorithme dynamique d'équilibrage de charge. L'indice de charge à équilibrer est le nombre de tâches en exécution sur chaque nœud. Dans notre implantation les tâches sont démarrées dès qu'elles deviennent locales et l'équilibrage est réalisé par un mécanisme de migration. L'algorithme que nous présentons se rapproche de celui de GTLB (cf. section 3.1.1).

Dans cet algorithme nous employons le module d'inspection de charge. Il est configuré pour permettre le contrôle du nombre de tâches en exécution dans un nœud (`CréationIndice`) : cet indice est incrémenté chaque fois qu'une tâche est démarrée (`AddV`) et decrémenté à chaque terminaison de tâche (`AddS`).

La politique que nous avons définie pour le déclenchement de l'envoi d'un message contenant l'information de charge d'un nœud prend en compte une variation de l'indice supérieure à un δ spécifié : δ ici représente un indice de surcharge. Nous avons aussi configuré un anneau pour les transmissions des informations de charge (`EnvoiIndice`). La réception d'un message implique l'exécution d'une fonction qui vérifie si le nœud est surchargé par rapport au nœud qui a envoyé le message (`RéceptionIndice`). Si c'est le cas une procédure de migration de tâches est activée à l'intérieur de la politique d'ordonnancement ; dans le cas contraire le message est envoyé au prochain nœud de l'anneau.

Ci-dessous nous identifions les opérations réalisées lors de l'appel aux services d'ordonnancement.

NouvelleTâche et NouvelÉtat
Toutes les tâches sont démarrées dès qu'elles deviennent locales. Lors de ce démarrage un nouveau processeur virtuel est créé pour son exécution (Créer_pv). Si la tâche est prête, le rapatriement de ses données est demandé (DéplacerDonnée). Les tâches qui ne sont ni prêtes ni locales sont ignorées.

ObtientTâche
Un appel à ce service est interprété comme la fin d'une tâche. Le processeur virtuel correspondant est détruit (Détruire_pv).

De plus une méthode interne à la politique est appelée par l'inspecteur de charge lorsque le nœud est considéré surchargé face à un autre. Ainsi cette méthode détermine le nombre de tâches qui doivent être migrées pour retrouver l'équilibre de charge entre les deux nœuds (ObtientIndice). Ensuite des tâches en nombre suffisant sont bloquées (PréempterTâche) et les processus virtuels correspondants détruits (Détruire_pv) ; les tâches bloquées sont alors envoyées au nœud sous-chargé (DéplacerTâche)[12].

8.3.2 Un algorithme de liste

Nous illustrons l'implantation d'un algorithme de partage de charge dynamique basé sur un algorithme de liste [65]. Le principe de cet algorithme est de maintenir une liste pour contenir l'ensemble de tâches prêtes produites par l'application ; ces tâches sont une à une exécutées par les processeurs. Dans notre implantation, cette liste est maintenue distribuée : chaque nœud comporte l'ensemble de tâches locales au nœud et le pool d'exécution consomme une à une les tâches de cette liste.

Dans la suite nous présentons les opérations exécutées pour chaque activation du noyau d'ordonnancement.

NouvelleTâche
Toute nouvelle tâche créée par l'application est soumise au noyau d'ordonnancement.

– Si la tâche n'est ni prête ni locale, elle est ignorée.
– Si la tâche est prête, cette politique demande le rapatriement sur le nœud des données requises par la tâche (service DéplacerDonnée).
– Si la tâche est locale, la tâche est additionnée à la liste de tâches.

[12]Il est à noter qui la préemption dans ATHAPASCAN-1, n'est possible que lorsque la tâche informé sa prédisposition à être preemptée.

`NouvelÉtat`
Chaque changement d'état des tâches est considéré par cette politique. Les opérations réalisées sont les mêmes que celles réalisées lors de la création d'une tâche.

`ObtientTâche`
Si la liste de tâches n'est pas vide, une tâche est retournée pour être exécutée par le pool d'exécution – la tâche choisie est celle qui a été plus récemment créée. Dans le cas contraire, un message contenant une demande de tâche est envoyée à un réplicat choisi arbitrairement.

Réception d'un message
La réception d'un message par un réplicat signifie la réception d'une demande de tâche. Si la liste est vide, le message est renvoyé à un autre réplicat ; dans le cas contraire la tâche la plus ancienne est envoyée au réplicat qui a demandé du travail en utilisant le service `DéplacerTâche` (les données de cette tâche sont aussi envoyées par l'intermédiaire du service `DéplacerDonnée`).

8.3.3 Un algorithme basé sur le vol de travail

Nous présentons dans cette section l'implantation d'un deuxième algorithme de partage de charge dynamique : un algorithme basé sur le « vol de travail ». Nous considérons un algorithme semblable à celui proposé par Cilk (section 3.2.2).

Dans notre implantation nous limitons la génération du parallélisme du programme aux situations dans lesquelles des ressources de calcul sont disponibles. Dans le cas où toutes les ressources de calcul (processeurs) sont occupées, les tâches sont exécutées séquentiellement : une création de tâche est convertie en un appel de fonction traditionnel.

Ci-dessous nous présentons les opérations réalisées par le noyau d'ordonnancement lors de l'appel à ces services.

`NouvelleTâche`
La génération du parallélisme est contrôlée lors de la création de tâches. Ainsi si un nœud est dans l'attente de service, c'est-à-dire qu'il n'a pas de tâches à exécuter, la tâche est créée et immédiatement migrée vers ce nœud (`DéplacerTâche` et `DéplacerDonnée`). Dans le cas contraire, la tâche n'est pas créée et le niveau applicatif doit l'exécuter sur le même flot d'exécution que la tâche créatrice (la valeur `faux` est retournée par `NouvelleTâche`, cf. section 8.1.1.1). Seules les

tâches locales sont concernées par ces opérations ; dans le cas de tâches prêtes, ces données en entrée sont rapatriées sur le nœud (`DéplacerDonnée`).

`NouvelÉtat`
Les tâches qui deviennent locales après leur création sont stockées dans une liste d'attente. Cette liste est en quelque sorte une réserve de travail futur sur le nœud. Dans le cas d'une tâche devenue prête, une demande de rapatriement de ses données sur le nœud (`DéplacerDonnée`) est réalisé.

`ObtientTâche`
Dès qu'une tâche se termine, le pool d'exécution demande la prochaine tâche pour l'exécuter. Cette tâche est recherchée dans la liste de tâches locales ; si la liste n'est pas vide, une tâche est alors retournée au pool d'exécution. Dans le cas contraire, un message de vol de travail est envoyé à un réplicat choisi aléatoirement.

Finalement, un réplicat qui reçoit un message de « vol » retourne au réplicat « voleur » une tâche de sa propre réserve. À défaut de tâches, le message est renvoyé à un autre réplicat. Le parcours de ce message est aléatoire, cependant après avoir parcouru $n - 1$ nœuds différents (n le nombre de nœuds de la machine) le message n'est plus renvoyé. Le dernier réplicat de la politique d'ordonnancement qui a reçu le message enregistre que le nœud qui a posté la requête de tâche est en attente de travail. Cette information sera prise en compte lors du prochain appel à `NouvelleTâche`.

8.3.4 Un algorithme statique

Nous présentons ici l'implantation d'une politique d'ordonnancement statique. Nous avons implanté[13] l'algorithme DSC que nous avons extrait du code source de PYRROS (section 3.3.1). Il est important de noter que cet algorithme nous fournit la distribution des tâches parmi les nœuds et l'ordre de leur exécution. Dans notre implantation, seule la distribution de tâches est considérée.

L'algorithme DSC prend en entrée une description d'un graphe de flot de données pour pouvoir calculer la distribution des tâches. Or dans ATHAPASCAN-1 la création de tâches est dynamique. Ainsi, pour appliquer DSC, le noyau d'ordonnancement attend que le graphe soit entièrement décrit : le noyau sait que le graphe est entièrement décrit lors d'un appel au service `GrapheDécrit`. Immédiatement le graphe d'ATHAPASCAN-1 est parcouru (par le service `LectureGraphe`) pour

[13] Travail réalisé avec Emmanuel Jeannot, LIP – Laboratoire d'Informatique du Parallélisme.

construire une représentation compatible avec celle requise en entrée par l'algorithme de DSC.

Dès que le placement du graphe a été calculé, il est appliqué par les services `DéplacerTâche` et `DéplacerDonnée`. Une nouvelle étape dans l'ordonnancement est initiée : celle de l'exécution.

L'algorithme de cette nouvelle étape est relativement simple : les tâches sont stockées dans une liste au fur et à mesure qu'elles deviennent locales (`NouvelÉtat`) et à chaque demande d'une tâche à exécuter (`ObtientTâche`) une tâche de cette liste est retournée – tant que cette liste n'est pas vide.

8.3.5 La spécialisation des stratégies

L'emploi d'une structure logiciel orientée objets permet la construction d'une hiérarchie de classes par spécialisation de fonctionnalités. Une nouvelle stratégie d'ordonnancement peut donc être implantée par la spécialisation d'une stratégie existante. Par exemple un algorithme glouton qui emploi un critère de priorité pour l'exécution des tâches peut être implanté à partir de la classe qui implante l'algorithme glouton présenté dans la section 8.3.2.

8.4 Conclusion

Dans ce chapitre nous avons présenté la réalisation d'un noyau d'ordonnancement pour l'environnement de programmation ATHAPASCAN. Il a été montré que ce noyau a été implanté de façon à supporter différentes politiques d'ordonnancement (section 8.1). Nous avons présenté une interface générique décrivant les points d'entrée aux services d'ordonnancement.

Nous avons aussi abordé le problème de la cohabitation de différentes politiques pour réaliser l'ordonnancement de tâches d'une application (section 8.2). Ce sujet a été abordé car, comme vu précédemment (dans la section 4.4), une application est composée d'une succession de phases de calcul. À chaque phase, une politique d'ordonnancement différente peut se montrer plus performante qu'une autre, d'où l'intérêt d'avoir des stratégies d'ordonnancement mixtes.

À la fin (section 8.3) nous avons présenté l'implantation de quatre politiques d'ordonnancement : trois dynamiques (deux algorithmes de partage de charge et un d'équilibrage de charge) et une statique (basé sur l'algorithme employé par DSC). L'implantation de ces quatre algorithmes illustre la capacité d'expression des algorithmes d'ordonnancement par la structure que nous avons proposée et construite.

Dans le chapitre suivant (chapitre 9) sont présentées quelques performances obtenues.

9

Évaluation du noyau d'ordonnancement

Nous présentons dans ce chapitre une évaluation quantitative du noyau d'ordonnancement implanté dans l'environnement de programmation parallèle ATHAPASCAN. Les point suivants sont traités :

1. Une application recursive, le problème des n-Reines. Nous abordons l'exécution de cette application sur trois architectures : sur une machine parallèle, sur un NOW et sur un SMP (section 9.2).
2. Une application numérique, la factorisation de Cholesky, sur une architecture parallèle (section 9.3).
3. L'emploi d'une stratégie mixte pour un problème qui possède un schéma similaire à une itération de Jacobi (section 9.4).
4. Une analyse des surcoûts du noyau d'ordonnancement par l'exécution d'un autre algorithme recursif, le calcul d'un terme de la suite de Fibonacci (section 9.5).
5. Une application de compression de fichier, `a1_gzip` (section 9.6).

Enfin, nous présentons dans la section 9.7 une visualisation *post-mortem* (obtenue par une trace) d'un intervalle de temps de l'exécution du noyau.

Dans la prochaine section nous présentons les conditions et les architectures sur lesquelles les expériences ont été réalisées.

9.1 Architectures utilisées et conditions d'expérimentation

Nous avons employé pour les expérimentations réalisées dans ce chapitre les architectures listées ci-dessous :

IBM-SP : architecture parallèle à mémoire distribuée, composée de 32 nœuds monoprocesseurs basés sur le processeur RS-6000/370, sous AIX 4.2.

NOW homogène : une architecture distribuée composée de 4 nœuds monoprocesseur Pentium 133, sous Solaris 2.6 et Ethernet 100 Mbps.

NOW hétérogène : une architecture distribuée composée de 10 nœuds (15 processeurs) basés sur des processeurs Pentium (1 nœud quadriprocesseur Pentium Pro 200, 2 nœuds biprocesseurs Pentium II 233, et 6 nœuds Pentium 133) sous Solaris Solaris 2.6 et Ethernet 100 Mbps.

SMP : le quadriprocesseur cité ci-dessus.

Tous les programmes utilisés ont été compilés avec des options d'optimisation (`-fast` sur les architectures basées sur de Pentium et `-O3` sur les RS-6000/370). Sur l'IBM-SP ATHAPASCAN-0 utilise la bibliothèque de processus légers POSIX et IBM-MPI pour les communications ; sur les configurations basées sur processeurs Pentium, ATHAPASCAN-0 utilise MPI-LAM et la bibliothèque de processus légers POSIX offerte par Solaris (*threads* système). Les mesures ont été prises lorsqu'aucune autre activité ne s'exécutait sur les machines. Tous les résultats présentés dans ce chapitre sont une moyenne des valeurs obtenues sur plusieurs exécutions, éliminant celles qui ont présentée une trop forte variation par rapport la moyenne (environ 10% de variance acceptée).

9.2 Une application récursive

L'application récursive considérée ici est le problème des n-Reines. Le but de ce problème est de à placer n reines sur un tableau d'échec de taille $n \times n$ de façon à ce qu'aucune reine ne soit en mesure d'en prendre une autre. L'algorithme implanté calcule le nombre de solutions valables ; les tâches sont créées récursivement pour exploiter un arbre de recherche. La donnée d'entrée de chaque tâche est l'échiquier avec les reines déjà placées. Le programme parallèle que nous avons construit prend en entrée le nombre n, définissant la taille du problème, et une valeur s, qui représente un seuil pour la génération de tâches à partir duquel l'exécution est faite

séquentiellement[1].

L'exécution de cette application a été soumise à un algorithme d'ordonnancement glouton, tel que celui présenté dans la section 8.3.2. La particularité de l'algorithme utilisé ici est de donner une plus haute priorité aux tâches les plus récentes (profondeur d'abord) lors d'une l'exécution locale et aux tâches les plus anciennes lors d'une migration.

Dans les tableaux 9.1 et 9.2 sont présentés les temps d'exécution de cette application pour n = 13, 14 et 15 et s = 3. Le nombre de tâches générées pour chaque taille du problème est aussi présenté. Dans ces deux tableaux, T_s correspond au temps d'une exécution séquentielle du programme sous l'environnement d'exécution ATHAPASCAN ; cela explique le gain de performance lors d'exécution parallèle sur un seul nœud par l'exploitation de la concurrence par les processeurs virtuels.

Tableau 9.1 *Le problème des n-Reines sur une architecture parallèle. Temps d'exécution du problème des n-Reines, pour trois tailles du problème, sur un IBM-SP. Temps en secondes.*

taille	# Tâches	T_s	1 nœud	2 nœuds	4 nœuds	8 nœuds	12 nœuds	16 nœuds
13	1178	36.66	38.04	20.03	9.28	5.06	2.92	2.40
14	1537	223.13	225.34	113.95	57.01	28.44	18.65	14.65
15	1964	1451.92	1450.74	779.39	364.84	188.41	130.80	91.60

Le tableau 9.1 présente les temps obtenus pour l'exécution du programme sur l'IBM-SP, utilisant de 1 à 16 nœuds monoprocesseur. Nous avons utilisé sur chaque nœud un pool d'exécution composé de 1 processeur virtuel. Les temps présentés nous permettent de calculer l'accélération[2] obtenue par l'exécution parallèle du problème. Cette accélération est donnée par $sp = T_s/T_p$, c'est-à-dire par le rapport entre le temps séquentiel et le temps de l'exécution parallèle sur p processeurs. Les valeurs de sp pour les temps du tableau 9.1 sont proches des valeurs optimales attendues, c'est-à-dire sp proche de p.

Tableau 9.2 *Le problème des n-Reines sur une architecture SMP et un NOW hétérogène. Temps d'exécution du problème des n-Reines pour trois tailles du problème. Temps en secondes.*

taille	# Tâches	SMP		NOW hétérogène $p(q)$: p nœuds et q processeurs							
		T_s	Parallèle	T_s	1(2)	2(4)	3(8)	4(9)	6(11)	8(13)	10(15)
13	1178	6.51	1.83	3.91	2.50	1.61	1.31	1.25	1.60	1.36	1.14
14	1537	39.50	11.40	23.78	14.19	8.37	6.60	6.25	5.02	4.84	4.96
15	1964	255.18	70.56	153.57	91.55	51.67	34.90	38.80	27.73	25.24	23.51

Le tableau 9.2 présente les temps mesurés pour cette application sur des architectures basées sur des processeurs Pentium : le SMP et le NOW hétérogène. Le

[1] Ce seuil défini en quelque sorte la granularité des tâches : plus grand est sa valeur, plus la granularité est grosse.

[2] Accélération ou en anglais *speed-up*.

pool d'exécution sur le SMP a été configuré avec 8 processeurs virtuels et chaque nœud du NOW dispose d'un pool d'exécution de 3 processeurs virtuels. Nous avons utilisé sept variations pour le NOW, variant le nombre de nœuds utilisés. Le temps séquentiel présenté a été mesuré sur le nœud le plus performant.

Nous pouvons observer par ces résultats que nous avons aussi obtenu une bonne exploitation du parallélisme de l'application. Cependant nous observons que les accélérations obtenues sur l'IBM-SP sont plus accentuées. Une des causes peut être le nombre de processeurs virtuels dans le pool d'exécution : nous avons utilisé un nombre fixe de processeurs virtuels dans le pool d'exécution de chaque nœud, sans considérer leur vitesse de calcul.

9.3 Une Application numérique

Nous considérons dans cette section un algorithme de factorisation de matrice symétrique : la factorisation de Cholesky[3]. Dans le programme implanté, les opérations sur des nombres scalaires ont été remplacées par des opérations sur des blocs : pour cela la matrice d'entrée est partitionnée en blocs de taille $(k \times k)$. Les opérations sur les blocs sont réalisées par des appels à la bibliothèque BLAS [42].

Dans le graphe de la figure 9.1 nous présentons la performance de l'exécution de ce programme sur 16 nœuds du IBM-SP, utilisant 1 processeur virtuel dans le pool d'exécution. La performance est présentée en Mflops (milliers d'opérations en nombre flottante par seconde), en sachant que la performance nominal théorique de chaque nœud est de 125 Mflops. Nous avons utilisé deux stratégies d'ordonnancement : un algorithme glouton (cf. section 8.3.2) et un placement cyclique bidimensionnel, dans laquelle une tâche qui actualise un bloc d'indices (i, j) est placée sur le nœud $(i \bmod q)q + (j \bmod q)$, où q^2 est le nombre de nœuds de la machine. La caractéristique de cette stratégie est de réduire les communications entre les nœuds.

Le graphe de la figure 9.1 montre que l'algorithme glouton n'est pas approprié au problème. Celui-ci ne tire pas parti de la localité des tâches et un grand volume de communications est observé pour actualiser les blocs de la matrice. En plus, de par la mauvaise distribution des données, la taille de la matrice est limitée. Cependant, même avec la stratégie cyclique bidimensionnelle nous n'obtenons pas de bonnes performances. Nous pouvons en partie attribuer cette mauvaise performance à une gestion dynamique d'un graphe construit par une application régulière.

Dans le cadre du travail de thèse de M. Doreille [43] un noyau d'ordonnancement spécialisé a été développé pour des problèmes sur lesquels un ordonnancement

[3] L'écriture de l'algorithme de Cholesky en ATHAPASCAN-1 et son ordonnancement sont traités par Doreille dans [43].

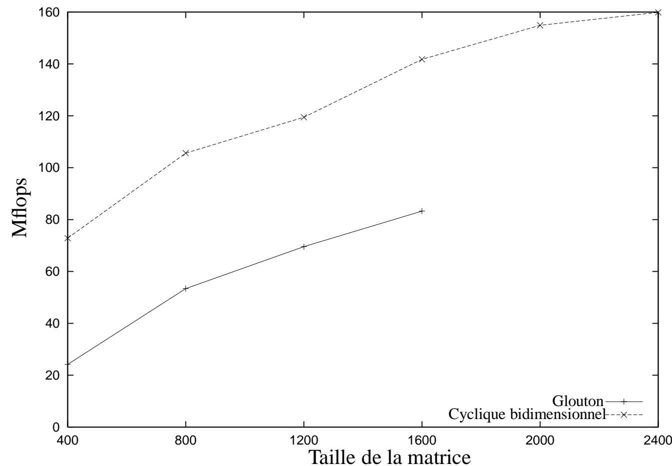

Figure 9.1 *Performance d'un algorithme de factorisation de Cholesky sur 16 nœuds d'un IBM-SP. Taille des blocs* $= 100$. *Deux politiques d'ordonnancement : gloutonne et bloc cyclique. Les courbes représentent le nombre de Mflops en fonction de la taille de la matrice.*

statique des tâches peut être calculé[4]. Un des points considérés lors du développement de ce noyau est de profiter de la connaissance complète du graphe pour calculer, en début d'exécution, un placement des tâches et ainsi réduire le surcoût de sa gestion dynamique.

Les performances obtenues avec le même programme de factorisation de Cholesky sur ce noyau sont dans le graphe de la figure 9.2. L'algorithme d'ordonnancement est le même algorithme cyclique bidimensionnel. Ces performances sont comparées à celles obtenues par ScaLAPACK [41], une bibliothèque spécialisée par le calcul numérique.

Malgré son caractère spécialisé en ce qui concerne le support de politiques d'ordonnancement statiques, le principe des interfaces a été maintenu. Ainsi le changement des stratégies d'ordonnancement reste possible. D'autres stratégies statiques ont été implantées sur ce noyau spécialisé, notamment un algorithme ETF (section 4.2.1.2) et DSC (section 4.2.1.3). Une analyse des performances obtenues est présentée dans [43].

[4]C'est le cas de la factorisation de Cholesky : le graphe de flot de données est entièrement décrit par la tâche racine du programme.

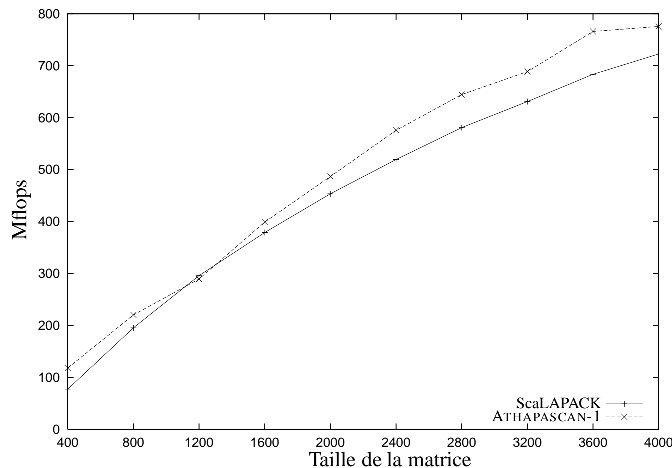

Figure 9.2 *Performance d'un algorithme de factorisation de Cholesky avec un noyau d'ordonnancement spécialisé. Taille des blocs* $= 100$*. Stratégie d'ordonnancement cyclique bidimensionnelle. Performance en Mflops de l'exécution sur 16 nœuds d'un IBM-SP ; performance de ScaLAPACK présenté pour la comparaison.*

9.4 Une stratégie mixte

Cette section présente une utilisation pratique d'une stratégie d'ordonnancement mixte. Nous appuyons notre discussion sur une application qui suit un schéma itératif de type Jacobi (cf. section 8.2).

L'algorithme implanté est présenté par l'algorithme 9.1. Dans cet algorithme un vecteur `v` de n éléments est créé ; chaque élément est soumis à un calcul itératif : ce calcul actualise une valeur `v[i]` prenant en compte `v[i-1]` et `v[i+1]`. Les données en entrée du programme sont la taille n du vecteur, qui correspond aussi au nombre de tâches créées à chaque itération, et le nombre it d'itérations à réaliser.

Les trois stratégies d'ordonnancement suivantes ont été utilisées pour l'exécution de cet algorithme (cf. discussion présentée dans la section 8.2).

- *Bloc cyclique* : cette stratégie réalise une distribution cyclique par bloc du vecteur de données. Ainsi chaque tâche qui manipule la donnée `v[i]` du vecteur est placée sur le processeur $\frac{i \times p}{n}$ (seule la partie entière est considérée), où p est le nombre de nœuds de la machine. Cette distribution est optimale pour réduire les communications entre les nœuds.
- *Glouton* : c'est l'algorithme présenté dans la section 8.3.2, déjà utilisé dans le problème des n-reines.
- *Mixte* : la distribution bloc cyclique est réalisée pour les tâches créées lors de la première itération. Pour les tâches créées à partir de la deuxième ité-

Algorithme 9.1 Un algorithme d'itération de Jacobi.

```
1:  Jacobi( int n, int it )
2:  double v[n]
3:  j = 1
4:  tant que j ≤ it faire
5:     v[1] = Calcul(v[1],v[2])
6:     i = 1
7:     tant que i ≤ n faire
8:        v[i] = Calcul(v[i-1],v[i],v[i+1])
9:        i++
10:    v[n] = Calcul(v[n-1],v[n])
11:    j++
```

ration est appliquée une stratégie gloutonne qui prend en compte la localité des données : les tâches sont placées sur le nœud où ses données d'entrée seront produites ; l'inactivité d'un nœud implique un processus de recherche de travail tel qu'utilisé par l'algorithme glouton.

Tableau 9.3 *Utilisation d'une stratégie d'ordonnancement mixte. Temps en secondes des exécutions de l'algorithme de Jacobi (algorithme 9.1) sur une architecture à 2 nœuds. Sont considérées deux situations, dans la première les deux nœuds sont dédiés à l'application, dans la deuxième une charge artificielle a été générée sur un des nœuds. Les lignes correspondent aux stratégies d'ordonnancement :* **B***loc* **C***yclique,* **G***loutonne et* **M***ixte. Les colonnes représentent diverses tailles du problème :* $it \times n$.

Stratégie	10×8	10×16	10×32	10×64	10×128
$2 \times$ monoprocesseur (dédiés)					
BC	1.78	3.30	6.45	12.73	24.05
G	2.33	3.86	7.82	15.51	32.03
M	1.80	3.49	6.32	13.22	25.42
$2 \times$ monoprocesseur (un nœud chargé artificiellement)					
BC	3.77	6.94	13.55	25.74	51.14
G	4.74	8.74	16.87	32.76	65.56
M	3.40	6.26	11.95	23.47	46.25

Les résultats obtenus avec ces trois stratégies sont présentés dans le tableau 9.3 (temps en secondes). Nous l'avons exécuté sur deux nœuds de la NOW homogène ; pool d'exécution de 2 processeurs virtuels. La taille d'un elément du vecteur est de 8 octets et le temps d'exécution sequentiel d'une tâche (un appel à la fonction `Calcul`) est de 200 μs. Nous avons considéré deux situations : dans la première

les deux nœuds sont entièrement dédiés à l'application, tandis que dans la deuxième une charge artificielle a été générée sur un des nœuds[5].

Les temps présentés montrent que la distribution bloc cyclique est en effet plus performante lorsque la machine est dédiée à l'application. Lorsqu'un nœud est plus rapide que l'autre, la stratégie mixte montre un bon compromis entre la localité et la vitesse des nœuds : elle permet après une bonne distribution initiale d'adapter le nombre de tâches à la vitesse de chacun des nœuds.

9.5 Les surcoûts du noyau d'ordonnancement

Nous présentons dans cette section une analyse des coûts liés à la gestion distribuée de l'exécution face à une exécution à plusieurs flots d'exécution obtenue par la variation du nombre de processeurs virtuels dans le pool d'exécution. Nous utilisons une application construite selon un algorithme de Fibonacci (algorithme 9.2) pour discuter ces coûts. Cette application implante un algorithme de découpe récursive qui prend en entrée une valeur n, représentant la taille du problème, et une valeur s, pour l'arrêt de la découpe (moment à partir duquel le calcul de Fibonacci est réalisé séquentiellement). La taille de s est directement liée à la granularité des tâches et inversement proportionnelle au nombre de tâches générées.

Algorithme 9.2 Un algorithme de Fibonacci.

```
1: RoutineParallèle Fibo( int n, int s, int& res)
2: si ( n ≤ s ) alors
3:    res = FibonacciSequentiel(n)
4: sinon {Exécuter en parallèle}
5:    int r1, r2
6:    Fibo( n-1, s, r1 )
7:    Fibo( n-2, s, r2 )
8:    res = r1 + r2
```

Dans le tableau 9.4 nous présentons les temps (en secondes) que nous avons mesurés pour l'algorithme 9.2 dans deux architectures monoprocesseur (un nœud du IBM-SP et un nœud de la NOW homogène) et sur trois architectures à 4 processeurs (4 nœuds IBM-SP, la NOW homogène et le SMP).

Nous avons exécuté cette application pour les entrées $n = 40$ et $s = 25$, en utilisant l'algorithme d'ordonnancement glouton (cf. section 8.3.2). Comme résul-

[5] La charge artificielle générée consommait environ 50% du temps du processeur ; valeur visualisée par l'utilitaire Unix `top`.

tat dans le tableau 9.4 nous présentons (*i*) le temps de l'exécution séquentielle de Fibonacci de 40, (*ii*) le temps de l'exécution séquentielle de Fibonacci de 25 et (*iii*) le temps d'exécution parallèle du programme suivant le nombre de processeurs virtuels dans le pool d'exécution de chaque nœud.

L'exécution parallèle du problème génère environs 3200 tâches, à peu près la moitié pour le calcul séquentiel de Fibonacci de 25. Ainsi approximativement la moitié des tâches s'exécutent en quelques microsecondes et les autres prennent quelques millisecondes (pour le calcul séquentiel de Fibonacci de 25).

Tableau 9.4 *Influence du nombre de flots d'exécution concurrents sur différentes architectures. Temps en secondes pour l'exécution d'un algorithme de Fibonacci sur 4 architectures : deux monoprocesseurs, une architecture parallèle, un NOW homogène et un SMP. Données en entrée pour l'exécution :* $n = 40$ *et* $s = 25$.

Nombre de processeurs virtuels	1 processeur		4 processeurs		
	RS-6000	*Pentium*	*IBM-SP*	*NOW*	*SMP*
Séquentiel(40)	43.43	36.09	—	—	21.78
Séquentiel(25)	0.031	0.026	—	—	0.017
1	89.47	63.69	25.09	19.14	22.72
2	66.68	51.41	18.74	14.70	11.37
4	55.36	44.02	16.05	12.51	5.75
8	50.58	40.43	14.19	11.49	5.71

9.5.1 Identification des surcoûts

Le surcoût total de l'exécution parallèle de l'algorithme 9.2 est donné par $\sigma = T_p - \frac{T_s}{p}$. Nous voulons dans cette section identifier les différentes sources de ce surcoût[6].

Nous avons employé le même algorithme pour le calcul du nombre de Fibonacci tant pour l'implantation du programme séquentiel que pour l'implantation du programme parallèle. Nous considérons ainsi qu'aucun surcoût n'a été introduit lors de l'expression du parallélisme du problème. Dans la version parallèle, les appels de fonction de l'algorithme 9.2 ont été remplacés par des constructeurs de tâches et la variable de résultat `res` a été placée dans une mémoire partagée (la variable d'écriture cumulative d'ATHAPASCAN-1 [59]).

Nous identifions trois sources de surcoûts :

[6]Notons que l'algorithme de Fibonacci utilisé est complètement parallélisable, ce qui facilite cette analyse.

Surcoût de gestion des processeurs virtuels et des communications. Ce surcoût est associé au support exécutif ATHAPASCAN-0. Nous observons son influence surtout dû à la présence du démon de communication : un processeur virtuel dédié au contrôle des communications. Son influence peut être calculée par le relation entre le temps d'exécution et le nombre de processeurs virtuels comme montre A. Carissimi dans [22]. Cette relation dit que si l'exécution d'un programme parallèle composé de q processeurs virtuels dans un nœud est réalisée dans un temps T, chacun des q processeurs virtuels a passé un temps d'environ T/q sur le processeur – considerant qui le travail est bien reparti entre les processeurs virtuels et que l'accès au processeur réel est equitable entre eux –.

Prenons comme exemple l'exécution à deux processeurs virtuels sur le RS-6000 : chaque processeur virtuel exécute approximativement la moitié du travail séquentiel, donc $43.43/2 = 21.72$ s. et le temps d'exécution attendu est de 2×21.72 plus le surcoût de gestion des deux processeurs virtuels sur le monoprocesseur. Considérant que le démon de communication utilise le processeur aussi longtemps que chacun des processeurs virtuels (donc $\frac{1}{3}$ du temps de processeur) nous avons $21.72 * 3 = 65.16$ s, proche du temps réel d'exécution qui est de 66.68 s et de la valeur de surcoût donnée par $\sigma = 23.25$ s.

Ce surcoût est masqué par l'addition de nouveaux processeurs virtuels (jusqu'à la capacité du nœud) participant au calcul ou par la présence de plusieurs processeurs, comme dans le cas de l'exécution sur un SMP.

Surcoût de génération et de contrôle du graphe. Ce surcoût est celui introduit par ATHAPASCAN-1 pour construire et gérer le graphe de flot de données associé à l'exécution du programme. Une analyse des différents composantes de ce coût est présentée par F. Galilée dans [59]. Il reflète fondamentalement les opérations d'addition de nouvelles tâches dans le graphe et la mise à jour des relations de précédence (la gestion de données) entre les tâches. Nous présentons dans la section 9.5.2 comment il est possible de diminuer le nombre d'opérations sur le graphe au niveau d'ordonnancement.

À partir des valeurs présentées dans le tableau 9.4 il est possible d'avoir une idée du surcoût de manipulation du graphe. Par exemple, l'exécution sur un monoprocesseur RS-6000 est réalisée dans un temps $T_1 = 89.47$ s en utilisant un seul processeur virtuel dans le pool d'exécution. Dû à la présence du démon de communication, le processeur virtuel a passé au moins $89.47/2 = 44.74$ s dans le processeur. La gestion du graphe a couté ainsi environ $44.74 - 43.43 = 1.32$ s.

Lors d'une exécution parallèle ce surcoût est plus élevé. Prenons l'exécution sur 4 nœuds, 1 processeur virtuel, sur l'IBM-SP : le temps T_4 est de 25.09 s. En sachant que $T_s = 43.43$ s et en considérant qu'une bonne distribution du travail a

été obtenue par l'algorithme glouton, chaque nœud a un travail équivalent à environ $43.43/4 = 10.86$ s. Or $T_4 = 25.09$ s. Si on considére toujours que la charge a été bien répartie, chaque nœud a travaillé (environ) 10.86 s pour l'application plus un temps égal pour le démon de communication, donc le temps d'exécution sur chaque nœud est de $2 \times 10.86 = 21.72$ s ; donc $25.09 - 21.72 = 3.37$ s par nœud a été consommé en opérations autres que l'exécution des tâches ou du démon. Cependant nous ne pouvons pas attribuer l'intégralité de ce surcoût à la gestion du graphe. En effet, une partie de ce surcoût vient du noyau d'ordonnancement.

Surcoût introduit par le noyau d'ordonnancement. Le surcoût introduit par le noyau d'ordonnancement est lié à la politique d'ordonnancement utilisée. Dans l'algorithme glouton, nous vérifions que ce surcoût se retrouve dans la gestion de la liste de tâches locales sur un nœud et, lors d'une exécution distribuée, de la gestion de la répartition des tâches parmi les nœuds. Dans la stratégie utilisée cette répartition est réalisée par des échanges de messages : (*i*) pour la demande de service, lorsqu'un nœud devient inactif, et (*ii*) pour la migration de tâches.

Tableau 9.5 *Opérations réalisées par un algorithme glouton. Comptabilisation du nombre de demandes de tâches, du nombre de tâches migrées et du nombre de tâches exécutées par nœud : sont présentés le nombre total de tâches exécutées et, entre parenthèses, le nombre de tâches calculant Fibonacci de 25. Les valeurs présentées correspondent à la trace de 5 exécutions de Fibonacci ($n = 40$ et $s = 25$) sur un NOW homogène à 4 monoprocesseurs, en utilisant un algorithme d'ordonnancement glouton et un pool d'exécution de 4 processeurs virtuels. Le nombre de tâches total généré par le programme est de 3192 (1595 pour le calcul de Fibonacci de 25).*

	# messages de vol				*# tâches migrées*				*# tâches exécutées*			
	1	2	3	4	1	2	3	4	1	2	3	4
1	5	7	8	6	13	3	5	4	803 (399)	813 (407)	800 (401)	777 (388)
2	8	9	11	9	14	9	7	5	813 (405)	781 (390)	815 (408)	783 (392)
3	7	6	6	4	8	4	5	5	755 (379)	783 (392)	834 (416)	820 (408)
4	5	5	6	4	12	2	1	4	765 (382)	772 (385)	842 (423)	813 (405)
5	4	8	6	7	14	2	4	4	802 (398)	805 (402)	805 (403)	780 (391)

Pour avoir une idée de la quantité de messages échangés au niveau d'ordonnancement, nous présentons dans le tableau 9.5 le nombre de messages de vols de travail envoyés par nœud et le nombre de tâches que le nœud a migrées. Les valeurs présentées ont été obtenues à partir de 5 exécutions du programme de Fibonacci ($n = 40$ et $s = 25$) utilisant l'ordonnancement glouton. Ces exécutions ont été réalisées sur le NOW homogène, avec 4 processeurs virtuels par pool d'exécution. Nous pouvons remarquer que le nombre total de vols de travail est considérablement inférieur au total de tâches créées par le programme.

Pour conclure l'analyse des surcoûts, nous résumons dans le tableau 9.6 les surcoûts mesurés pour l'exécution parallèle de Fibonacci ($n = 40$ et $s = 25$) sur des architectures à mémoire distribuée ; les surcoûts σ sont présentés en pourcentage de T_p. Nous avons utilisé l'IBM-SP et le NOW homogène en utilisant 2, 3 et 4 nœuds pour analyser l'influence du nombre de nœuds dans le surcoût. Nous pouvons vérifier par ces résultats que le surcoût tend à diminuer avec l'ajout de ressources de calcul (processeurs virtuels ou nœuds). En ce qui concerne l'efficacité ($E = \frac{T_s/T_p}{p}$) de l'exécution parallèle, nous remarquons qu'elle décroît légèrement avec le rajout de nœuds et qu'elle augmente avec le nombre de processeurs virtuels.

Tableau 9.6 *Surcoûts d'une exécution distribuée. Les résultats montrent l'évolution du surcoût ($\sigma_{\%}$) et l'efficacité (E) de l'exécution parallèle d'un algorithme de découpe recursive sur deux architectures parallèles en variant les ressources de calcul : processeurs virtuels (# pv) et nœuds.*

#pv	2 nœuds				3 nœuds				4 nœuds			
	IBM-SP		NOW		IBM-SP		NOW		IBM-SP		NOW	
	$\sigma_{\%}$	E	$\sigma_{\%}$	E	$\sigma_{\%}$	E	$\sigma_{\%}$	E	$\sigma_{\%}$	E	$\sigma_{\%}$	E
1	61.42	0.45	49.00	0.51	42.93	0.44	35.26	0.48	32.77	0.43	28.03	0.47
2	32.80	0.60	27.33	0.64	24.12	0.58	19.45	0.63	18.15	0.58	15.73	0.61
4	19.21	0.72	16.17	0.75	14.61	0.70	12.14	0.73	11.96	0.68	9.66	0.72
8	12.63	0.80	10.76	0.82	9.63	0.78	8.56	0.80	7.67	0.77	6.84	0.79

Dans la section suivante nous allons analyser en quoi une politique d'ordonnancement peut réduire le surcoût d'exécution d'un programme ATHAPASCAN-1.

9.5.2 Influence de l'ordonnancement dans le surcoût d'exécution d'un programme ATHAPASCAN-1

Une partie du surcoût d'exécution est rajoutée par la manipulation du graphe par ATHAPASCAN-1, notamment lors de la création des tâches. Or selon [14] ce surcoût peut être réduit par l'emploi d'une technique de « vol de travail », présentée à la section 8.3.3. Ainsi de nouvelles tâches sont créées seulement lorsqu'un processeur devient inactif ; avant cela, toutes les tâches sont exécutées comme des appels à procédure, selon l'ordre d'exécution donné par le programme séquentiel.

Un algorithme de vol de travail a été utilisé pour ordonnancer l'algorithme de Fibonacci sur un SMP. Nous considérons que le nombre de tâches créées dans le programme est considérablement plus important que le nombre de vols de travail (cela a été vérifié dans le tableau 9.5), ainsi une tâche n'est rajoutée dans ce graphe que lorsqu'un processeur est inactif.

Le tableau 9.7 présente les résultats obtenus par l'exécution d'un algorithme de Fibonacci sur le SMP (pool d'exécution composé de 4 processeurs virtuels, un par

Tableau 9.7 *Réduction du surcoût de manipulation du graphe par le vol de travail. Temps en secondes pour l'exécution d'un algorithme de Fibonacci pour differentes tailles du problème sur un SMP : exécution séquentielle et en parallèle supportée par un algorithme glouton et par un algorithme basé sur le vol de travail.*

taille	# Tâches	*Séquentiel*		*Glouton*		*Vol de travail*	
		T_s	$\frac{T_s}{\text{\# tâches}}$	T_p	$\frac{T_p}{\text{\# tâches}}$	T_p	$\frac{T_p}{\text{\# tâches}}$
35	18454860	$1980e^{-3}$	$0.107e^{-6}$	1309.61	$71.0e^{-6}$	$850e^{-3}$	$0.046e^{-6}$
30	1664064	$170e^{-3}$	$0.102e^{-6}$	120.3	$72.3e^{-6}$	$80e^{-3}$	$0.048e^{-6}$
25	150048	$17e^{-3}$	$0.113e^{-6}$	10.89	$72.6e^{-6}$	$12e^{-3}$	$0.08e^{-6}$
20	13528	$2e^{-3}$	$0.148e^{-6}$	0.97	$71.7e^{-6}$	$3e^{-3}$	$0.22e^{-6}$
15	1218	$0.8e^{-3}$	$0.657e^{-6}$	0.08	$65.7e^{-6}$	$3e^{-3}$	$2.46e^{-6}$
10	108	$0.7e^{-3}$	$6.48e^{-6}$	0.01	$92.6e^{-6}$	$3e^{-3}$	$27.8e^{-6}$

processeur réel) ordonnancé par une politique de vol de travail. La première colonne présente les tailles considérées (le nombre pour lequel est calculé le numéro de Fibonacci), la deuxième présente le nombre de tâches créées lors d'une exécution parallèle (ou d'appels de fonction dans l'exécution séquentielle). Les colonnes contiennent les temps et les rapports entre le temps et le nombre de tâches obtenu par l'exécution du programme de Fibonacci en séquentiel et avec les deux politiques d'ordonnancement suivantes.

1. *Glouton* : l'exécution parallèle est supportée par la politique d'ordonnancement de liste utilisée pour les mesures présentées dans le tableau 9.4. La différence étant qu'ici nous utilisons un seuil $s = 2$, donc sans contrôle du grain au niveau applicatif.
2. *Vol de travail* : l'exécution parallèle supportée par une politique de vol de travail, donc avec un contrôle de grain au niveau de l'ordonnancement.

Comme montré par les résultats du tableau 9.7, la stratégie de vol de travail n'est pas seulement plus performante sur le SMP que l'algorithme glouton mais aussi plus performante que l'exécution séquentielle. Cela est dû au fait que les 4 processeurs disponibles dans la machine sont exploités pour exécuter une partie du calcul.

9.6 Compression de fichiers

Nous présentons dans cette section la parallélisation d'une application de compression de fichiers : `gzip`. Cette application a été programme en ATHAPASCAN-1 par B. Carton et F. Giquel. Le rapport [23] contient une description complète des résultats.

Le point de départ de cette parallélisation est le code séquentiel de `gzip` (et aussi de `gunzip`). Une rapide encapsulation en C++ a permis de le rendre exécutable en tant que programme ATHAPASCAN-1. `gzip` possède la particularité de pouvoir décompresser des fichiers concaténés. La parallélisation de la compression reposait donc sur une technique de type « diviser pour reigner » : le fichier en entrée était découpé en blocs, puis chaque bloc était compressé en parallèle puis les résultats de ces compressions étaient concaténés afin d'obtenir la compression du fichier total.

En pratique, le programme `a1_gzip` est implanté par un ensemble de tâches indépendantes, chaque tâche prenant en entrée une portion d'un fichier (déjà stocké en mémoire) et offrant en sortie une zone de mémoire contenant la portion du fichier compressée. D'autres tâches prennent en charge la lecture du fichier source et l'écriture du fichier sortie.

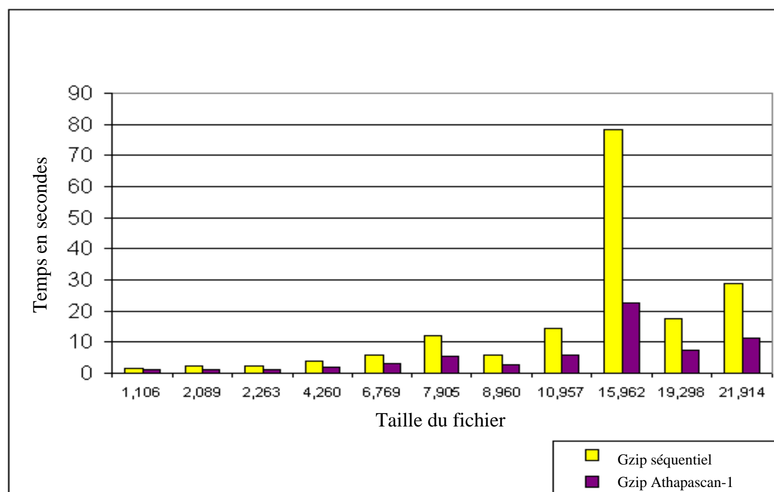

Figure 9.3 *Performance d'un algorithme de compression de fichiers. Temps en secondes pour l'exécution d'un programme de compression de fichiers appliquée à de fichiers de différents tailles.*

Le graphe de la figure 9.3 présente les temps (en secondes) obtenues pour l'exécution d'`a1_gzip` pour la compression de différents fichiers de différents tailles (et aussi différentes natures : images, exécutables, données, *etc*.). L'algorithme d'ordonnancement utilisé a été un algorithme de liste centralisé. Les temps correspondants à l'exécution séquentielle de `gzip` sont aussi présentés. Ces temps correspondent à l'exécution sur le SMP, pool d'exécution de 4 processeurs virtuels.

9.7 Visualisation d'une politique d'ordonnancement

Au sein du projet ATHAPASCAN nous avons à disposition **Pajé** [112, 113], un outil de visualisation *post-mortem* de l'exécution de programmes parallèles, tout spécialement de programmes ATHAPASCAN-0. Comme le noyau d'ordonnancement décrit dans ce document a été construit sur ATHAPASCAN-0, cet outil peut être employé directement. Nous pouvons ainsi observer par la trace toutes les opérations réalisées sur ATHAPASCAN-0, comme par exemple, la prise d'un verrou, l'envoi d'un message sur le réseau et si un processeurs virtuel est bloqué sur une primitive de synchronisation ou s'il est en cours d'exécution.

Nous avons profité de la structure modulaire de Pajé pour ajouter de nouvelles informations de visualisation qui permettent de suivre le comportement du noyau d'ordonnancement.

Dans la figure 9.4 est présentée la visualisation d'un intervalle de temps dans l'exécution du noyau d'ordonnancement. Dans cette *trace* nous pouvons vérifier dans les trois traits supérieurs les proportions de temps ou chaque processeur virtuel exécute des activités utilisateur et du noyau d'ordonnancement[7]. Les flèches correspondent à la manipulation des tâches : les créations et les démarrages. Dans la version actuelle de Pajé il n'a pas été possible d'identifier individuellement les tâches, ainsi est seulement réalisée la comptabilisation du nombre de tâches dans chaque état (prête ou non prête).

9.8 Bilan des résultats

Nous avons présenté dans ce chapitre une analyse quantitative du noyau d'ordonnancement applicatif que nous avons implanté pour l'environnement de programmation ATHAPASCAN. Dans cette analyse nous sommes particulièrement intéressés à l'ordonnancement dynamique. Dans [29] nous présentons une comparaison entre un ordonnancement cyclique bidimensionnel et l'algorithme de DSC appliqués à un algorithme d'élimination de Gauss.

Dans les sections 9.2 et 9.3 nous avons présenté une analyse de performance

[7]Malheureusement en dégradé de gris l'identification de ces états n'est pas si simple. La couleur foncée représente les activités d'ordonnancement – et l'initialisation d'un processeur virtuel – et la couleur claire l'exécution d'une tâche utilisateur. Toujours à cause de l'absence de couleurs, nous ne représentons pas les états correspondant aux opérations propres à ATHAPASCAN-1, nous les considérons comme des opérations réalisées par les tâches.

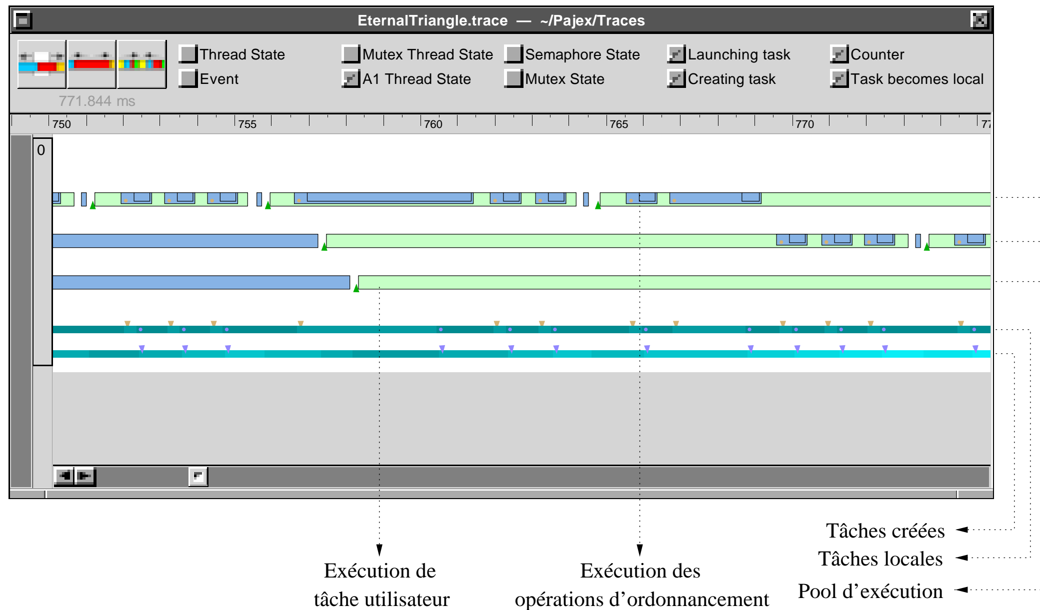

Figure 9.4 *Visualisation de l'exécution du noyau d'ordonnancement. Nous visualisons le comportement pendant un intervale de temps d'une exécution d'une politique sur le noyau d'ordonnancement. Les trois traits supérieurs correspondent aux trois processeurs virtuels créés dans le pool d'exécution ; une couleur foncée indique que le processeur virtuel est en train d'exécuter une opération d'ordonnancement, une couleur claire indique qu'il est en train d'exécuter une tâche utilisateur, l'absence de couleur indique que le processeur virtuel est inactif. Les deux traits inférieurs correspondent à des compteurs du nombre de tâches présentes sur le nœud, le trait d'en haut correspond au nombre de tâches créées non prêtes et le trait d'en bas au nombre de tâches prêtes non lancées. Les flèches montrent les variations d'état subies par les tâches : une flèche d'un processeur virtuel vers le compteur de tâches non prêtes indique une création de tâche et une flèche partant du compteur de tâches prêtes vers un processeur virtuel, le démarrage d'une tâche. Les démons* ATHAPASCAN-0 *ne sont pas montrés.*

du noyau d'ordonnancement appliqué à un algorithme de génération recursive de tâches (le problème des n-reines) et à un algorithme numérique qui manipule une grande quantité de données (la factorisation de Cholesky). Les performances présentées montrent que le noyau d'ordonnancement proposé est assez efficace pour des applications générant un graphe de flot de données à la volée. Nous avons aussi montré que ce noyau peut être spécialisé pour implanter des stratégies d'ordonnancement ciblées à des classes d'applications dans le cas considéré des programmes générant un graphe régulier, comme montré pour l'algorithme de factorisation de Cholesky.

Dans la section 9.4 nous avons présenté une utilisation pratique d'une stratégie d'ordonnancement mixte. Nous avons utilisé deux politiques pour contrôler l'exécution d'un algorithme qui suit le schéma itératif de Jacobi : un bloc cyclique pour distribuer les données et un algorithme glouton pour réguler la charge entre les nœuds de la machine.

Dans la section 9.5 nous avons analysé les surcoûts liés à l'exécution d'un programme. Nous avons identifié trois sources de surcoût :

1. Gestion au niveau applicatif du graphe de flot de données.
2. Gestion au niveau exécutif des processeurs virtuels et des communications.
3. Gestion de la répartition de la charge de l'application entre les nœuds de la machine. Nous avons vu que ce surcoût dépend de la politique d'ordonnancement utilisée.

Dans la section 9.5.2 nous avons présenté comment le noyau d'ordonnancement peut, par l'utilisation d'une stratégie basée dans le vol de travail, réduire le surcoût de gestion du graphe d'un programme ATHAPASCAN-1.

Enfin nous avons présenté dans la section 9.7 une visualisation *post-mortem* d'un intervalle de temps de l'exécution du noyau d'ordonnancement.

10

Conclusion

Dans le cadre du projet APACHE, nous avons au cours de cette thèse spécifié et implanté un noyau d'ordonnancement applicatif pour l'environnement de programmation ATHAPASCAN. Dans ce contexte, l'intérêt d'un tel noyau est d'offrir une couche de portabilité et d'efficacité aux programmes parallèles.

Nous montrons dans cette thèse que l'ordonnancement peut être profondément séparé tant du niveau applicatif que de l'architecture. À partir de cette constatation nous avons proposé un module d'ordonnancement indépendant, permettant que la conception d'un programme parallèle soit considérée séparément de son support d'exécution.

En outre nous proposons une librairie permettant l'implantation de différentes politiques d'ordonnancement. Ainsi la stratégie d'ordonnancement d'une application peut être facilement adaptée lorsqu'on change de machine cible.

Travaux réalisés

Le travail développé dans cette thèse a été entièrement intégré à la définition et à l'implantation de l'interface de programmation ATHAPASCAN-1, supervisée par Jean-Louis Roch, en étroite liaison avec les travaux de thèse de Mathias Doreille [43] et François Galilée [59].

En ce qui concerne le document présenté les points suivants ont été abordés :

– La séparation du noyau d'ordonnancement du niveau applicatif et de l'archi-

tecture par une interface d'échanges de services.

- L'identification d'un graphe de flot de données macroscopique et dynamique comme élément de synthèse des interactions entre un algorithme d'ordonnancement d'une part et un programme parallèle d'autre part.
- La définition de la structure interne d'un noyau d'ordonnancement prévoyant un mécanisme de changement de politique d'ordonnancement.
- L'implantation d'un noyau d'ordonnancement pour l'environnement de programmation parallèle ATHAPASCAN.
- L'implantation sur ce noyau de différentes stratégies d'ordonnancement et leur évaluation sur des applications programmées en ATHAPASCAN-1.

L'ensemble des résultats est concrétisé par une bibliothèque, fournie avec l'interface applicative ATHAPASCAN-1, implantant le noyau d'ordonnancement.

Travaux futurs

Nous ne considérons pas le travail présenté dans cette thèse comme terminé. Au contraire, nous pensons que la recherche dans le domaine peut être poursuivie. Une suite naturelle est l'utilisation du noyau d'ordonnancement pour l'implantation de nouvelles stratégies d'ordonnancement. Cependant d'autres axes de recherche sont ouverts. Nous en citons trois.

Le noyau d'ordonnancement a été défini et implanté comme un module totalement indépendant du niveau applicatif. Cependant nous avons concentré nos efforts de mise au point pour des programmes ATHAPASCAN-1. Or dans le projet APACHE, un certain nombre d'applications ne sont pas développées sur ATHAPASCAN-1 (par exemple la décomposition de domaines pour la simulation numérique [32] et la simulation de circulation océanique [10]). La tâche de programmation de telles applications pourrait être réduite en utilisant le noyau d'ordonnancement présenté dans cette thèse comme un outil de virtualisation de l'architecture.

Nous avons vu dans le chapitre 9, lors de l'évaluation de performance du noyau, que dans certains cas l'utilisation d'un noyau spécialisé peut se montrer plus intéressant qu'un noyau générique, notamment sur des applications régulières (sections 9.3 et 9.5.2). Or il est tout à fait envisageable de réunir les caractéristiques d'un tel noyau spécialisé et de celles du noyau générique que nous proposons au sein d'un seul module, de façon à s'adapter à l'irrégularité de l'application au cours de son exécution.

Finalement nous envisageons le développement d'un noyau d'ordonnancement pour un langage orienté objets. Un tel noyau pourrait réaliser la régulation de charge

à deux niveaux : à gros grain pour contrôler la distribution des objets et à grain plus fin, lors du contrôle de l'exécution des méthodes. Ce travail est l'articulation de deux travaux de recherche auxquels j'ai déjà participé : DPC++ pour la partie distribution d'objets et ATHAPASCAN-1 pour la partie régulation des appels de méthodes.

Annexe A

La classe A1_DECISION

Nous présentons ici la classe A1_DECISION. Dans cette classe nous avons implanté l'interface du noyau d'ordonnancement avec ATHAPASCAN-1 et avec le niveau exécutif. Notons que cette classe n'implante aucune politique d'ordonnancement. L'implantation d'une politique d'ordonnancement consiste en *spécialiser*, dans le sens C++, cette classe par une autre et implanter la fonctionnalité désirée aux méthodes spécifiées. Notons aussi qu'une grande partie de nos efforts lors de l'implantation de notre environnement d'exécution parallèle, tout spécialement en ce qui concerne le noyau d'ordonnancement, a été dédiée à l'ordonnancement de tâches. Ainsi nous présentons l'interface entre le noyau d'ordonnancement et ATHAPASCAN-1 qui permet de suivre l'évolution du graphe en fonction des changements d'états des tâches.

La partie du code que nous présentons ci-dessous correspond à la définition de la classe A1_DECISION. Dans cette transcription, nous avons mis en caractères *gras* le nom des méthodes. Notons néanmoins qu'une politique d'ordonnancement peut ne pas avoir besoin de toutes les fonctionalités définies dans cette classe. Cependant, deux méthodes doivent nécessairement être définies : ***local_closure*** et ***get_closure_to_run***.

```
class a1_decision {
public:
  // Les fonctions ci-dessous correspondent à l'interface du
  // noyau d'ordonnancement avec Athapascan-1

  /** Fonction appelée lors de la création locale d'une
      clôture par la clôture mère. */
  virtual void new_closure( a1_closure& mere,
                            a1_closure& cree,
                            a1_state::state s ) {}
```

```
  /** Fonction appelée lors que un graphe est déjà
      complètement décrit */
  virtual void group_flush() {}

  /** Fonction appelée lors d'une réception d'une clôture
      par le réseau. */
  virtual void network_closure( a1_closure& n_closure,
                                a1_state::state s ) = 0;

  /** Fonction appelée pour une nouvelle clôture prête.
      (pas de prédécesseur) */
  virtual void ready_closure( a1_closure& r_closure ) {}

  /** Fonction appelée pour une nouvelle clôture locale
     (données locales). */
  virtual void local_closure( a1_closure& l_closure ) = 0;

  // Les fonctions ci-dessous correspondent à l'interface du
  // noyau d'ordonnancement avec le niveau exécutif

  /** Fonction appelée lors qu'un processeur virtuel cherche
      une nouvelle clôture pour exécuter. Si aucune clôture
      est local la fonction doit retourner 0 (zéro). */
  virtual a1_closure* get_closure_to_run() = 0;

  /** Fonction lors du début d'exécution d'une clôture */
  void begin_run_closure( a1_closure& j ) {}

  /** Fonction lors de la fin d'exécution d'une clôture */
  void end_run_closure( a1_closure& j ) {}

  /** Retour: priorité de la politique d'ordonnancement.
      Par défaut toutes politiques ont la même priorité. */
  virtual int priority() { return 0; }
};
```

Annexe B

L'ordonnancement dans un programme ATHAPASCAN-1

Nous présentons dans cet annexe l'utilisation du noyau d'ordonnancement à l'intérieur d'un programme ATHAPASCAN-1 (cf. syntaxe présentée dans la section 7.1.4). Nous renvoyons le lecteur à la thèse de M. Doreille [43] pour plus d'informations sur la syntaxe d'ATHAPASCAN-1.

Le programme présentée calcule un terme de la suite de Fibonacci à partir d'une entrée `n`.

Un exemple d'une création de tâche spécifiant un stratégie d'ordonnancement différente de celle utilisée par défaut est présenté dans la ligne 24. Les autres créations de tâches (lignes 16 à 18 et 15) les tâches emploient la stratégie d'ordonnancement défaut.

Les informations données aux tâches `Fibo` dans **`SchedAttributs(n-1)`** correspond au coût d'exécution de la tâche.

Program B.1 Calcul de Fibonacci en ATHAPASCAN-1.

```
 1: struct PrintRes {
 2:     void operator()(Shared_r<int> res ) {
 3:         cout « "Res = " « res.read() « endl ;
 4:     }
 5: }
 6: struct Somme {
 7:     void operator()(Shared_r<int> r1, Shared_r<int> r2, Shared_cw res ) {
 8:         res.cumul( r1.read() + r2.read() ) ;
 9:     }
10: }
11: struct Fibo {
12:     void operator()(int n, Shared_cw<int> res ) {
13:         if( n <= 2 ) res.cumul( 1 ) ;
14:         else {
15:             Shared<int> r1, r2 ;
16:             Fork<Fibo>(SchedAttributs(n-1))(n-1, r1)) ;
17:             Fork<Fibo>(SchedAttributs(n-2))(n-2, r2)) ;
18:             Fork<Somme>()(r1, r2, res) ;
19:         }
20:     }
21: }
22: int main( int argc, char** argv ) {
23:     Shared<int> res ;
24:     Fork<Fibo>(glouton, SchedAttributs(atoi(argv[1])))(atoi(argv[1]), res)) ;
25:     Fork<PrintRes>()(res) ;
26: }
```

Bibliographie

[1] American National Standards Institute. – *IEEE standard for information technology : Portable Operating Sytem Interface (POSIX). Part 1, system application program interface (API) — amendment 1 — realtime extension [C language]*. – 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, IEEE Computer Society Press, 1994, xxiii + 590p. IEEE Std 1003.1b-1993 (formerly known as IEEE P1003.4 ; includes IEEE Std 1003.1-1990). Approved September 15, 1993, IEEE Standards Board. Approved April 14, 1994, American National Standards Institute.

[2] Arvind et Culler (D. E.). – Resource requirements of dataflow programs. *In : Proceedings of the 15th Annual International Symposium on Computer Architectures*, pp. 141–150. – Honolulu, Hawai, May 1988.

[3] Ávila (R. B.). – *Um Modelo de Paralelismo de Grão Fino para Objetos Distribuídos*. – Porto Alegre, Brazil, Thèse de master, CPGCC, Federal University of Rio Grande do Sul, 1999. In Portuguese, to be published.

[4] Barreto (M. E.), Navaux (P. O. A.) et Rivière (M. P.). – Deck : Distributed executive communication kernel. *In : IV CACiC - Congresso Argentino de Ciencia de la Computacion*. – Neuquen, Argentina, octobre 1998.

[5] Bernard (G.) et Folliot (B.). – Caractéristiques générales du placement dynamique : synthèse et problématique. *In : École française de parallélisme, réseaux et systemes*, pp. 3–25. – INRIA, 1996.

[6] Bernard (G.), Stève (D.) et Simatic (M.). – Placement et migration de processus dans les systèmes réparties faiblement couplés. *Technique et Sciences Informatiques*, vol. 10, n5, mai 1991, pp. 375–392.

[7] Bernard (P.-E.). – *Parallélisation et multiprogrammation pour une application irrégulière de dynamique moléculaire opérationnelle*. – Grenoble, France, Thèse de doctorat, Institut National Polytechnique de Grenoble, octobre 1997.

[8] Birrell (A. D.) et Nelson (B. J.). – Implementing remote procedure calls. *ACM Transactions on Computer Systems*, vol. 2, n1, février 1984, pp. 39–59.

[9] Black (D. L.). – Scheduling support for concurrency and parallelism in the Mach OS. *IEEE Computer*, vol. 23, n5, avril 1990, pp. 35–43.

[10] Blayo (E.), Debreu (L.), Mounié (G.) et Trystram (D.). – Dynamic load balancing for ocean circulation model with adaptive meshing. *In : Europar'99*. – à paraître.

[11] Blazewicz (J.), Ecker (K.), Schmidt (G.) et Weglarz (J.). – *Scheduling in Computer and Manufacturing Systems*. – Springer-Verlag, 1993.

[12] Blelloch (G. E.). – Programming parallel algorithms. *Communications of the ACM*, vol. 39, n3, mars 1996, pp. 85–97.

[13] Blumofe (R. D.). – *Executing Multithreaded Programs Efficiently*. – Massachusetts, Thèse de doctorat, DEECS Massachusetts Institut of Technology, septembre 1995.

[14] Blumofe (R. D.), Joerg (C. F.), Kuszmaul (B. C.), Leiserson (C. E.), Randall (K. H.) et Zhou (Y. C. E.). – Cilk : an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, vol. 30, n8, août 1995, pp. 207–216.

[15] Blumofe (R. D.) et Leiserson (C. E.). – Space-efficient scheduling of multithreaded computations. *In : Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing (STOC '93)*, pp. 362–371. – San Diego, CA, USA, mai 1993. Also submitted to SIAM Journal on Computing.

[16] Blumofe (R. D.) et Leiserson (C. E.). – Scheduling multithreaded computations by work stealing. *In : Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, éd. par Goldwasser (Shafi). pp. 356–368. – Los Alamitos, CA, USA, novembre 1994.

[17] Booch (G.). – *Objet-Oriented Analysis & Design*. – New York, Benjamin-Cummings, 1993, 2ème édition.

[18] Braschi (B.). – *Principes de base des algorithmes d'ordonnancement de liste et affectation de priorites aux tâches*. – Grenoble, France, Thèse de doctorat, Institut National Polytechnique de Grenoble, septembre 1988.

[19] Briat (J.), Gautier (T.) et Roch (J.-L.). – Application irrégulier et ordonnancement en ligne. *In : École française de parallélisme, réseaux et systemes*, pp. 81–105. – INRIA, 1996.

[20] Briat (J.), Ginzburg (I.), Pasin (M.) et Plateau (B.). – Athapascan runtime : Efficiency for irregular problems. *In : Proceedings of the Europar'97 Conference*. pp. 590–599. – Passau, Germany, août 1997.

[21] Canin (J.). – Tera Computer attempts to beat the odds. *Supercomputing Review*, vol. 5, n5, mai 1992.

[22] Carissimi (A. S.). – *Le noyau exécutif Athapascan-0 et l'exploitation de la multiprogrammation légère sur les grappes de stations multiprocesseurs.* – Grenoble, France, Thèse de doctorat, Institut National Polytechnique de Grenoble, septembre 1999.

[23] Carton (B.) et Giquel (F.). – *Data Processing using* ATHAPASCAN-1. – 3rd year engineering school project report, ENSIMAG, Grenoble, juin 1999.

[24] Casavant (T. L.) et Kuhl (J. G.). – A taxonomy of scheduling general-purpose distributed computing systems. *IEEE Trans. on Software Engineering*, vol. 14, n2, février 1988, pp. 141–154.

[25] Castañeda-Retiz (M. R.), Rouchon (P.) et Schiavo (O.). – Étude quantitative des stratégies pour la régulation dynamique de charge. *In : École française de parallélisme, réseaux et systemes*, pp. 283–299. – INRIA, 1996.

[26] Cavalheiro (G. G. H.). – *Um Modelo para Linguagens Orientadas a Objetos Distribuído*. – Porto Alegre, Brésil, Thèse de master, CPGCC–UFRGS, août 1994.

[27] Cavalheiro (G. G. H.), Denneulin (Y.) et Roch (J.-L.). – A general modular specification for distributed schedulers. *In : Proceedings of Europar'98*, éd. par Springer Verlag (LNCS 980). – Southampton, England, septembre 1998.

[28] Cavalheiro (G. G. H.) et Doreille (M.). – *Athapascan* : A C++ library for parallel programming. *In : Proceedings of the Stratagem'96*. INRIA, p. 75. – Sophia Antipolis, France, Juillet 1996.

[29] Cavalheiro (G. G. H.), Doreille (M.), Galilée (F.), Gautier (T.) et Roch (J.-L.). – Athapascan-1 : A multithreaded execution model based on data flow. Soumis.

[30] Cavalheiro (G. G. H.), Krug (R. C.), Rigo (S. J.) et Navaux (P. O. A.). – DPC++ : An object–oriented distributed language. *In : XV SCCC*. – Arica, Chile, novembre 1995.

[31] Cavalheiro (G. G. H.) et Roch (J.-L.). – Un schéma modulaire pour l'Écriture des ordonnanceurs. *In : Actes du RenPar'98*. – Strasbourg, France, juin 1998.

[32] Charão (A. S.), Charpentier (I.) et Plateau (B.). – A framework for parallel multithreaded implementation of domain decomposition methods. *In : Proceedings of Parallel Computing'99 (to be published)*.

[33] Chretienne (Ph.), Coffman, Jr. (E. G.), Lenstra (J. K.) et Liu (Z.) (édité par). – *Scheduling Theory and Its Applications*. – Chichester, England, Wiley, 1995.

[34] Coffman, Jr. (E. G.) et Graham (R. L.). – Optimal scheduling for two-processor systems. *Acta Informatica*, no1, 1972, pp. 200–213.

[35] Cohen (W. E.), Yalamanchili (N.), Tewari (R.), Patel (C.) et Kazi (T.). – Exploitation of multithreading to improve program performance. *In : Proceedings of The Yale Multithreaded Programming Workshop*. – New Haven, EUA, juin 1998.

[36] Colin (J.-Y.) et Chrétienne (P.). – CPM scheduling with small interprocessor communication delays. *Operations Research*, vol. 39, n3, 1991.

[37] Cung (V.-D.), Fraigniaud (P.), Gautier (T.) et Trystram (D.). – De l'algorithme au support. *In : ICaRE'97 : conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*, éd. par Barth (D.), Chassin de Kergommeaux (J.), Roch (J.-L.) et Roman (J.). – Aussois, France, décembre 1997.

[38] Denneulin (Y.). – *Conception et ordonnancement des applications hautement irrégulières dans un contexte de parallélisme à grain fin*. – Lille, France, Thèse de doctorat, LIFL – Université des Sciences et Technologies de Lille, octobre 1998.

[39] Denneulin (Y.), Namyst (R.) et Méhaut (J. F.). – Architecture virtualization with mobile threads. *In : Parallel Computing : Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, 19-22 September 1997, Bonn, Germany*, éd. par D'Hollander (E. H.), Joubert (G. R.), Peters (F. J.) et Trottenberg (U.). pp. 477–484. – Amsterdam, février 1998.

[40] Dijkstra (E. W.). – Cooperating sequential processes. *In : Programming Languages*, éd. par Genuys (F.), pp. 43–112. – Academic Press, 1968.

[41] Dongarra (J. J.) et Walker (D. W.). – Software libraries for linear algebra computations on high performance computers. *SIAM Review*, vol. 37, n2, juin 1995, pp. 151–180.

[42] Dongarra (J.J.), Du Croz (J.), Hammarling (S.) et Duff (I.). – A Set of Level-3 Basic Linear Algebra Subprograms. *ACM Trans.Math.Software*, vol. 16, 1990, pp. 1–17,18–28.

[43] Doreille (M.). – *Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. – Grenoble, France, Thèse de doctorat, Institut National Polytechnique de Grenoble, décembre 1999.

[44] Eager (D. L.), Lazowska (E. D.) et Zahorjan (J.). – A load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, vol. 12, n15, mai 1986, pp. 662–675.

[45] El-Rewini (H.), Lewis (T. G.) et Ali (H. H.). – *Task Scheduling in Parallel and Distributed Systems*. – Englewood Cliffs, Prentice Hall, 1994.

[46] Elsner (U.). – *Graph Partitioning : A Survey*. – Rapport technique n SFB393/97-27, Chemnitz, Fakultaet fuer Mathematik, TU Chemnitz, décembre 1997.

[47] Feitelson (D.). – *A survey of scheduling in multiprogrammed parallel systems*. – Rapport technique n 19790(87657), Chemnitz, IBM T. J. Watson Research Center, octobre 1994. Revision d'août 1997.

[48] Ferrari (D.). – *A Study of Load Indices for Load Balancing*. – Technical Report n CSD-86-262, University of California, Berkeley, octobre 1985.

[49] Ferrari (D.) et Zhou (S.). – A load indices for dynamic load balancing. *In : Proc. Fall Join Computer Conf*, pp. 684–690. – Dallas, Texas, 1986.

[50] Feschet (F.), Miguet (S.) et Perroton (L.). – ParList : Une structure de données parallèle pour l'équilibrage des charges. *In : Parallèlisme et applications irrégulières*, éd. par Authié (G. et al.), chap. 8, pp. 71–87. – Hermès, 1995.

[51] Flynn (M. J.). – Some computer organizations and their effectiveness. *IEEE Trans. Computers*, vol. C-21, n9, septembre 1972, pp. 948–960.

[52] Folliot (B.). – *Méthodes et outils de partage de charge pour la conception et la mise en œuvre d'applications dans les systèmes repartis hétérogènes*. – Paris, Thèse de PhD, Université Paris VI, Institut Blaise Pascal, 1992.

[53] Folliot (B.). – *Contribution à une approche système du placement dynamique dans les systèmes répartis hétérogènes*. – Paris, Habilitation à diriger des recherches, Université Pierre et Marie-Curie – Paris VI, 1996.

[54] Fonlupt (C.). – *Distribuition dynamique de données sur machines SIMD*. – Lille, France, Thèse de doctorat, LIFL – Université des Sciences et Technologies de Lille, janvier 1997.

[55] Foster (I.), Kesselman (C.) et Tuecke (S.). – The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, vol. 37, n1, août 1996, pp. 70–82.

[56] Fu (C.) et Yang (T.). – Run-time compilation for parallel sparse matrix computations. *In : FCRC'96 : Conference proceedings of the 1996 International Conference on Supercomputing : Philadelphia, Pennsylvania, USA, May 25–28, 1996*, éd. par ACM. pp. 237–244. – New York, USA, 1996.

[57] Galilée (F.) et Roch (J.-L.). – Langages pour l'expression dynamique de parallélisme et graphe de tâches. *In : ICaRE'97 : conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*, éd. par Barth (D.), Chassin de Kergommeaux (J.), Roch (J.-L.) et Roman (J.). – Aussois, France, CNRS, décembre 1997.

[58] Galilée (F.), Roch (J.-L.), Cavalheiro (G. G. H.) et Doreille (M.). – Athapascan-1 : On-line building data flow graph in a parallel language. *In : Pact'98*. – Paris, France, octobre 1998.

[59] Galilee (F.). – ATHAPASCAN-*1 : Interprétation distribuée du flot de données d'un programme parallèle*. – Grenoble, France, Thèse de doctorat, Institut National Polytechnique de Grenoble, septembre 1999.

[60] Gautier (T.), Roch (J.-L.) et Villard (G.). – Regular versus irregular problems and algorithms. *In : Proc. of IRREGULAR'95*. – Lyon, France, Septembre 1995.

[61] Geib (J. M.), Denneulin (Y.), Namyst (R.) et Ménhaut (J.-F.). – Processus légers distribués et régulation de charge. *In : École française de parallélisme, réseaux et systemes*, pp. 153–170. – INRIA, 1996.

[62] Geist (A.), Beguelin (A.), Dongarra (J.), Jiang (W.), Manchek (R.) et Sunderam (V.). – *PVM : Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. – Cambridge, Massachusetts, The MIT Press, 1994.

[63] Gerasoulis (A.) et Yang (T.). – On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, n6, juin 1993, pp. 686–701.

[64] Ginzburg (I.). – *Athapascan-0b : intégration efficace et portable de multiprogrammation légère et de communications*. – Grenoble, France, Thèse de doctorat, Institut National Polytechnique de Grenoble, septembre 1997.

[65] Graham (R. L.). – Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, vol. 17, n2, mars 1969, pp. 416–429.

[66] Guyennet (H.) et Philippe (L.). – Application aux systèmes à architecture parallèle et répartie. *In : École française de parallélisme, réseaux et systemes*. pp. 27–44. – INRIA.

[67] Hafidi (Z.). – *MARS : un environnement de programmation parallèle adaptative dans les réseaux de machines hétérogènes multi-utilisateurs*. – Lille, France, Thèse de doctorat, LIFL – Université des Sciences et Technologies de Lille, septembre 1998.

[68] Hafidi (Z.) et E-G. Talbi (J-M. Geib). – MARS : Un ordonnanceur adaptatif d'applications parallèles dans un environnement multi-utilisateurs. *In : Actes du RenPar'96*. – Bordeaux, France, mai 1996.

[69] Halstead (R. H.). – Multilisp : A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, vol. 7, n 4, octobre 1985, pp. 501–538.

[70] Hwang (J. J.), Chow (Y.-C.), Anger (F. D.) et Lee (C.-Y.). – Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, vol. 18, n2, avril 1989, pp. 244–257.

[71] Hwang (K.) et Briggs (F. A.). – *Computer Architecture and Parallel Processing*. – McGraw-Hill International, 1985.

[72] Itzkovitz (A.), Schuster (A.) et Shalev (L.). – *Millipede : Towards standard interface for virtual parallel machines on top of distributed environments*. – Rapport technique nLPCR-9607, Haifa, Computer Science Department, Israel Institute of Technology, janvier 1996.

[73] Jacqmot (C.). – *Load Management in Distributed Computing Systems : Towards Adapti ve Strategies*. – Louvain, Belgium, Thèse de doctorat, Université Catholique de Louvain, FSA–DII, janvier 1996.

[74] Joerg (C.). – *The Cilk system for parallel multithreading computing*. – Massachusetts, Thèse de doctorat, DEECS Massachusetts Institut of Technology, janvier 1996.

[75] Karp (R. K.), Luby (M.) et Meyer auf der Heide (F.). – Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, vol. 16, n4/5, octobre/novembre 1996, pp. 517–542.

[76] Karypis (G.) et Kumar (V.). – Analysis of multilevel graph partitioning. *In : Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, éd. par Karin (Sidney). – New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995.

[77] Karypis (G.) et Kumar (V.). – Parallel multilevel k-way partitioning scheme for irregular graphs. *In : Supercomputing '96 Conference Proceedings : November 17–22, Pittsburgh, PA*, éd. par ACM. – New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996.

[78] Karypis (G.) et Kumar (V.). – A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, vol. 20, n1, janvier 1999, pp. 359–392.

[79] Kazman (R.), Bass (L.), Abowd (G.) et Webb (M.). – SAAM : A method for analyzing the properties of software architectures. *In : Proceedings of the 16th International Conference on Software Engineering*. pp. 81–90. – Sorrento, Italy, mai 1994.

[80] Konig (J.-C.) et Roch (J.-L.). – Machines virtuelles et techniques d'ordonnancement. *In : ICaRE'97 : conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*, éd. par Barth (D.), Chassin de Kergommeaux (J.), Roch (J.-L.) et Roman (J.). – Aussois, France, décembre 1997.

[81] Kumar (A.), Singhal (M.) et Ming (T.). – A model for distributed decision making : an expert system for load balancing in distributed systems. *In : 11th Symp. Operating System*. pp. 507–513. – IEEE.

[82] Lam (S.) et Sethi (R.). – Worst case analysis of two scheduling algorithms. *SIAM Journal on Computing*, vol. 6, n3, septembre 1977, p. 518.

[83] Lin (F. C. H.) et Keller (R. M.). – The gradient model load balancing method. *IEEE Transactions on Software Engineering*, vol. 13, n1, janvier 1987, pp. 32–38.

[84] Lisper (B.). – Detecting static algorithms by partial evaluation. *In : Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*. pp. 31–42. – New York, 1991.

[85] Liu (Z.). – A note on Graham's bound. *Information Processing Letters*, vol. 36, n1, octobre 1990, pp. 1–5.

[86] Lowenthal (D. K.), Freeh (V. W.) et Andrews (G. R.). – Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing, Special Issue on Multithreading for Multiprocessors*. – À apparaître.

[87] Markatos (E. P.). – How architecture evolution influences the scheduling discipline used in shared-memory multiprocessors. *In : Parallel Computing (ParCO) '93*. pp. 7–10. – Grenoble, France, 1993.

[88] Message-Passing Interface Forum. – *MPI-2.0 : Extensions to the Message-Passing Interface*. – MPI Forum, juin 1997.

[89] Namyst (R.). – *PM^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. – Lille, France, Thèse de doctorat, LIFL – Université des Sciences et Technologies de Lille, janvier 1997.

[90] Narlikar (G. J.) et Blelloch (G. E.). – Space-efficient implementation of nested parallelism. *In : Proceedings of the ACM SIGPLAN Symposium on Principles and Practice od Parallel Programming (PPOPP-97)*. pp. 25–36. – New York, juin 18–21 1997.

[91] Narlikar (G. J.) et Blelloch (G. E.). – Pthreads for dynamic and irregular parallelism. *In : Proceedings of High Performance Network and Computing*. – Orlando, USA, novembre 1998.

[92] Ni (L. M.), Xu (C.-W.) et Gendreau (T. B.). – Distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, vol. SE-11, n10, octobre 1985, pp. 1153–1161.

[93] Nikhil (R. S.). – Parallel symbolic computing in Cid. *In : Workshop on Parallel Symbolic Computing*. pp. 217–242. – Beaune, France, octobre 1995.

[94] Orlando (S.) et Perego (R.). – Scheduling data-parallel computations on heterogeneous and time-shared environments. *In : Proceedings of Europar'98*, éd. par Springer Verlag (LNCS 980). – Southampton, England, septembre 1998.

[95] Pasin (M.). – *Mouvement efficace de donnees pour la programmation parallele irreguliere*. – Grenoble, France, Thèse de doctorat, Institut National Polytechnique de Grenoble, octobre 1999.

[96] Pellegrini (F.) et Roman (J.). – SCOTCH : A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *In : Proceedings of HPCN'96*. – Brussels, Belgium, 1996. LNCS 1067.

[97] Perrin (G.-R.) et Darte (A.). – *The data parallel programming model : foundations, HPF realization, and scientific applications*. – New York, NY, USA, Springer-Verlag Inc., 1996, *Lecture Notes in Computer Science*, volume 1132, xv + 284p.

[98] Pilla (M.), Barreto (M. E.), Santos (R. R.), Cavalheiro (G. G. H.) et Navaux (P. O. A.). – Implementacao de um algoritmo de criação de checkpoints para a linguagem distribuída DPC++. *In : Proceedings of the II CACIC'97*. – La-Plata, Argentina, 1997.

[99] Plateau (B.). – *Présentation d'APACHE*. – Rapport APACHE n 1, Grenoble, IMAG, octobre 1993.

[100] Randall (K. H.). – *Cilk : Efficient Multithreaded Computing*. – Massachusetts, Thèse de doctorat, DEECS Massachusetts Institut of Technology, juin 1998.

[101] Rinard (M. C.) et Lam (M. S.). – The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, vol. 20, n3, mai 1998, pp. 483–545.

[102] Rinard (M. C.), Scales (D. J.) et Lam (M. S.). – Jade : A high-level machine-independent language for parallel programming. *Computer*, vol. 26, n6, juin 1993, pp. 28–38.

[103] Roch (J.-L.). – Parallel efficient algorithms and their programming. – 1997. Polycopié ENSIMAG, Grenoble, France – Première édition adaptée d'un tutorial à ISSAC'97 –.

[104] Roch (J.-L.) et Villard (G.). – *Parallel Computer Algebra*. – Rapport technique, Lecture notes for a tutorial in ISSAC'97, Hawaii, Jul 1997.

[105] Roch (J.-L.), Villard (G.) et Roucairol (C.). – Algorithmes irréguliers et ordonnancement. *In : Parallèlisme et applications irrégulières*, éd. par Authié (G. et al.), chap. 4, pp. 71–87. – Hermès, 1995.

[106] Sarkar (V.). – *Partitioning and Scheduling Parallel Programs for Multiprocessors*. – The MIT Press, 1989.

[107] Schuster (A.) et Shalev (L.). – *Access hisories : how to use the princiep of locality in distributed shared memory systems*. – Rapport technique nLPCR-9701, Haifa, Computer Science Department, Israel Institute of Technology, janvier 1997.

[108] Shmoys (D. B.), Wein (J.) et Williamson (D. P.). – Scheduling parallel machines on-line. *SIAM Journal on Computing*, vol. 24, n6, décembre 1995, pp. 1313–1331.

[109] Skillicorn (D. B.) et Talia (D.). – Models and languages for parallel computation. *ACM Computing Surveys*, vol. 30, n2, juin 1998, pp. 123–169.

[110] Snir (M.), Otto (S. W.), Huss-Lederman (S.), Walker (D. W.) et Dongarra (J.). – *MPI : the complete reference*. – Cambridge, MA, USA, MIT Press, 1996, xii + 336p.

[111] Stankovic (J. A.). – Simulations of three adaptive, decentralized controlled job scheduling algorithm. *Computer Network*, vol. 8, août 1984, pp. 199–217.

[112] Stein (B. O.). – *Visualisation interactive et extensible de programmes parallèles à base de processus légers*. – Grenoble, France, Thèse de doctorat, Université Joseph Fourier, octobre 1999.

[113] Stein (B. O.) et Chassin de Kergommeaux (J.). – Interactive visualisation environment of multi-threaded parallel programs. *In : Parallel Computing : Fundamentals, Applications and New Directions*. pp. 311–318. – Elsevier.

[114] Stenström (P.). – VLSI Support for a Cactus Stack Oriented Memory Organization. *In : Proc. of the 21st Annual Hawaii International Conference on System Sciences. Vol I Architecture Track*, éd. par Hoevel (L. W.). pp. 211–220. – Hawaii, janvier 1988.

[115] Talbi (E. G.). – Un algoritmes d'allocation dynamique de processus sur un réseau de Transputers. *La Lettre du Transputer*, vol. 11, septembre 1991, pp. 7–20.

[116] Talbi (E.-G.). – *Allocation dynamique de processus dans les systèmes distribués et parallèles : État de l'art*. – Rapport technique nTR-162, LIFL, Université de Lille 1, janvier 1995.

[117] Talbi (E. G.). – Une taxonomie des algoritmes d'allocation dynamique de processus dans les systèmes parallèles et distribués. *In : ISYPAR'97 2ème Ecole d'Informatique des Systèmes Parallèles et Répartis*, pp. 137–164. – Toulouse, France, mars 1997.

[118] Tanenbaum (A. S.). – *Modern Operating Systems*. – New Jersey, Prentice Hall, 1992.

[119] Tarnvik (E.). – Dynamo – a portable tool for dynamic load balancing on distributed memory multicomputers. *In : CONPAR'92*, éd. par Bouge (L.), Cosnard (M.), Robert (Y.) et Trystram (D.). pp. 485–490. – Lyon, France, septembre 1992. LNCS 634.

[120] Trémollet (Y.). – *Parallélisation d'algorithmes variationnels d'asimilation de données en météorologie*. – Grenoble, France, Thèse de doctorat, Université Joseph Fourier, octobre 1997.

[121] Valiant (L. G.). – A bridging model for parallel computation. *Communications of the ACM*, vol. 33, n8, août 1990, pp. 103–111.

[122] Valiant (L. G.). – General purpose parallel architectures. *In : Handbook of Theoretical Computer Science – Algorithms and Complexity)*, éd. par van Leeuwen (J.). – Elsevier and MIT Press, 1990.

[123] van der Steen (A. J.) et Dongarra (J. J.). – *Overview of Recent Supercomputers*. – Rapport technique nUT-CS-96-325, Department of Computer Science, University of Tennessee, avril 1996.

[124] Verriet (J.). – *Scheduling with communication for multiprocessor computation*. – Utrecht, the Netherlands, Thèse de doctorat, Utrecht University, Department of Computer Science, juin 1998.

[125] Watts (J.). – *A Practical Approach to Dynamic Load Balancing*. – Passadena, USA, Thèse de master, California Institute of Technology, octobre 1995.

[126] Watts (J.), Rieffel (M.) et Taylor (S.). – Practical dynamic load balancing for irregular problems. *In : Proceedings of IRREGULAR'96*, pp. 299–306. – Santa Barbara, USA, août 1996. LNCS 1117.

[127] Willebeek-LeMair (M. H.) et Reeves (A. P.). – Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, n9, septembre 1993, pp. 979–993.

[128] Yang (T.) et Fu (C.). – Space/time-efficient scheduling and execution of parallel irregular computations. *ACM Transactions on Programming Languages and Systems*, vol. 20, n6, novembre 1998, pp. 1195–1222.

[129] Yang (T.) et Gerasoulis (A.). – A fast static scheduling algorithm for DAGs on a unbounded number of processors. *In : Proceedings of the 4th Annual Conference on Supercomputing*, éd. par MacCallum (Anne Copeland). pp. 633–640. – Alburquerque, NM, USA, novembre 1991.

[130] Yang (T.) et Gerasoulis (A.). – Pyrros : Static task scheduling and code generation for message passing multiprocessors. *In : 6th ACM International Conference on Supercomputing*, pp. 428–437. – Washington, D.C., juillet 1992.

[131] Zhou (S.). – *Performance Studies of Dynamic Load Balancing in Distributed Systems*. – Technical Report n CSD-87-376, University of California, Berkeley, 1987.

Résumé

Dans les environnements d'exécution parallèle, la régulation de charge (ou l'ordonnancement applicatif) est le module responsable du contrôle de l'exécution d'un programme sur les ressources de l'architecture distribuée (processeurs et modules mémoire). En pratique, le choix de la stratégie de régulation la plus performante dépend non seulement de l'application mais doit aussi être adapté en fonction de l'architecture cible. Dès lors, la portabilité d'un code ne peut être assurée que si l'on peut modifier cette stratégie.
Dans cette thèse, nous proposons l'utilisation de la description dynamique du flot de données comme l'élément central permettant de séparer le code applicatif de la régulation de charge. Sur cette proposition est basée la construction d'un environnement logiciel, modulaire et générique, qui rend possible la modification ou l'ajustement de la stratégie de régulation de charge.
La spécification de cet environnement repose sur l'identification des interfaces de la régulation avec d'une part l'application et d'autre part l'architecture. Cette identification, centrée sur l'exploration macroscopique du flot de données, est originale : nous montrons qu'elle étend d'autres systèmes classiques de régulation de charge.
Enfin, la validation expérimentale de cet environnement est réalisée grâce à son intégration dans l'interface de programmation ATHAPASCAN-1 de l'environnement ATHAPASCAN, du projet APACHE. Différentes stratégies d'ordonnancement, statiques, dynamiques et hybrides, ont ainsi été implantés. Nous présentons les performances de quelques unes de ces stratégies appliquées à des programmes ATHAPASCAN-1 sur différentes architectures.

Mots-clés : Architectures parallèles et distribuées, Programmation parallèle, Ordonnancement, Régulation de charge dynamique, Flot de données.

ATHAPASCAN-1 *: Generic scheduling interface in a parallel execution environment*

Abstract

In a parallel programming environment, the load sharing module – or application level scheduler – manages the execution of a program over the resources of a distributed architecture: processors and memory modules. In practice, the efficiency of the scheduling strategy highly depends on the characteristics of both application and target architecture. Consequently, to achieve code portability, it should be possible to tune the scheduling strategy.
In this thesis, we propose the use of a dynamic data flow graph as the key element in order to separate the application code from its scheduling strategy. This approach allows the construction of a modular and generic software environment that enables to tune the scheduling strategy.
The specification of this environment relies on the identification of interfaces between the load sharing module and both the application and the architecture. This approach, based on a macroscopic analysis of the data flow graph, is original: we show that it can be seen as an extension of other traditional load balancing systems.
Finally, an experimental validation of this environment was carried out through its implementation in ATHAPASCAN-1. ATHAPASCAN-1 is the programming interface of the ATHAPASCAN parallel programming environment developed in the APACHE project. Various scheduling strategies (static, dynamic and hybrid) have been implemented. We present the performance of some of these strategies applied to several programs on various architectures.

Keywords: Parallel and distributed architectures, Parallel programming, Parallel environments, Scheduling, On-line load sharing, Data-flow.