2m11.2577.4

Université de Montréal

Conception et réalisation d'une variante parallèle de C basée sur la création paresseuse de tâche

par

Francis L'Écuyer

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

Décembre 1997

©Francis L'Écuyer, 1997

QA
76
U54
1998
v.007

Université de Montréal
Faculté des études supérieurs

Ce mémoire intitulé:

Conception et réalisation d'une variante parallèle du langage C basée sur la création paresseuse de tâche

présenté par:

Francis L'Écuyer

a été évalué par un jury composé des personnes suivantes:

Paul Bratley - Président du jury

Marc Feeley - Directeur de recherche

Paola Flocchini - Membre du jury

Mémoire accepté le: 5 décembre 1997

Sommaire

Conception et réalisation d'une variante parallèle de C basée sur la création paresseuse de tâche

Mémoire présenté à la Faculté des études supérieures de l'Université de Montréal.

par Francis L'Écuyer

Ce mémoire porte sur la conception d'une variante parallèle du langage C, appelée ParSubC, conçue pour exprimer du parallélisme de contrôle. La motivation principale pour la conception de cette nouvelle variante est la spécialité et la complexité qui caractérisent souvent les variantes parallèles existantes du C. ParSubC, qui a comme but d'être une variante simple et générale, n'est en fait qu'une simple extension du langage C (un seul nouvel opérateur pour créer des tâches).

Des compilateurs ParSubC ont été implantés sur deux types d'environnement parallèle: un ordinateur multi-processeur à mémoire partagée et un multi-ordinateur constitué d'un réseau de stations de travail. Les deux implantations sont basées sur une technique de partitionnement de tâches appelée *Création paresseuse de tâche*. L'idée principale de la CPT est de retarder le plus possible la création d'une tâche, c'est-à-dire tant qu'aucun processeur n'en a besoin, dans le but de réduire le surcoût associé au parallélisme.

Un aspect important de l'implantation de ParSubC sur le multi-ordinateur est la présence d'une mémoire partagée virtuelle, qui offre au programmeur un modèle de programmation à mémoire partagée (malgré que la mémoire physique soit possiblement distribuée). Mise-à-part la présence de deux nouveaux qualificatifs permettant l'identification des variables partagées, l'utilisation de la mémoire partagée virtuelle est aussi simple que celle d'une mémoire partagée physique.

En terminant, pour avoir une idée des performances possibles avec la variante ParSubC, plusieurs programmes ParSubC ont été écrits et exécutés dans les deux environnements parallèles. Différentes mesures provenant de l'exécution de ces programmes ont étés prises pour permettre d'évaluer le langage ParSubC et ses implantations.

Remerciements

J'aimerais remercier ma famille ainsi que mes amis pour leur support et leur encouragement tout au long de la rédaction de ce mémoire.

J'aimerais également remercier mon directeur de maîtrise, Marc Feeley, pour sa patience et son assistance tout au long de ce projet.

Un merci tout spécial à Anne-Marie Favreau, ma conjointe, pour sa patience et sa compréhension pour toutes ces fins de semaine et soirées passées à rédiger.

Pour terminer, j'aimerais également remercier le Centre de Recherche Informatique de Montréal (CRIM) pour leur support financier ainsi que Nortel pour son équipement informatique avec lequel j'ai pu évaluer une implantation de ParSubC.

Table des matières

1 Introduction **1**

1.1 Motivation 1

1.2 Langages destinés au parallélisme de contrôle 2

1.2.1 Concurrent C 2

1.2.2 IRIS Power-C 5

1.3 Architectures 7

1.3.1 Cray T3D et réseau de stations HP 8

1.4 Modèle de programmation à mémoire partagée 8

1.4.1 Mémoire partagée virtuelle 10

1.5 Code généré 10

2 Concepts de programmation parallèle **14**

2.1 Caractéristiques d'un langage parallèle 15

2.1.1 Disponibilité du langage 15

2.1.2 Gestion des tâches 16

2.1.3 Modèle de programmation 16

2.2 Modèle de programmation de ParSubC 17

2.2.1 Modèle "fork-join" 18

2.2.2 Appel de fonction de style "Remote procedure call" 18
2.2.3 La récursivité 19
2.2.4 Diviser-pour-régner 20
2.3 Discussion sur ParSubC 21
2.3.1 Simplicité 21
2.3.2 Souplesse 21
2.3.3 Puissance 22
2.3.4 Résumé 23

3 La création paresseuse de tâche 24

3.1 Principes de la CPT 24
3.1.1 Vol de tâche 24
3.1.2 Création de tâche 26
3.1.3 Ordonnancement des tâches 26
3.1.4 Suspension de tâche 27
3.2 Protocoles d'accès aux piles 28
3.2.1 Protocole à mémoire partagée 28
3.2.2 Protocole à envoi de messages 29
3.2.3 Protocole présenté dans ce chapitre 29
3.3 Implantation de la CPT 30
3.3.1 Gestion et exécution des tâches 30
3.3.2 Recherche de travail 37
3.4 Résumé 40

4 Implantation avec mémoire distribuée 42

4.1 Notation utilisée dans ce chapitre 42

4.2 Architecture de l'implantation 43

4.3 Programme ParSubC 46

4.4 Communications 46

4.4.1 Interface au réseau 46

4.4.2 Distributeur de messages 49

4.5 Mémoire partagée virtuelle 51

4.5.1 Variables partagées: copie unique *vs* copies multiples 52

4.5.2 Messages et opérations de la MPV 53

4.5.3 Organisation de la mémoire partagée virtuelle 56

4.5.4 Identification des variables partagées 58

4.5.5 Migration de tâche et variables locales 60

4.5.6 Cohérence des variables globales 62

4.6 Partitionnement CPT 63

4.6.1 Migration de tâche 64

4.6.2 Ordonnancement et migration de tâche 67

4.7 Résumé 71

5 Implantation avec mémoire partagée **73**

5.1 Mémoire partagée du CRAY T3D 73

5.1.1 Accès à la mémoire partagée 74

5.1.2 Variables globales 75

5.2 Partitionnement 76

5.2.1 Protocole à mémoire partagée *vs* à envoi de messages 76

5.2.2 Techniques d'interruptions pour le PEM 78

5.3 CPT basée sur le PEM pour mémoire partagée 82

5.3.1 Envoi d'une requête de vol 83

5.3.2 Détection de la réception d'une requête de vol 85

5.3.3 Traitement d'une requête 87

5.4 Résumé 88

6 Évaluation **89**

6.1 Mesures et méthodes d'évaluation 89

6.1.1 Loi d'Amdahl 91

6.2 Performances du Cray T3D 91

6.3 Performances avec le réseau de stations HP 95

6.3.1 Programmes avec migration de tâches 97

6.3.2 Migration de données *vs* migration de tâches 103

6.4 Discussion sur les implantations 104

6.4.1 Partitionnement de tâches 104

6.4.2 Facteur de dérangement des processeurs libres 107

6.4.3 Coût du "polling" avec le Cray T3D 108

6.4.4 Mémoire partagée 109

6.4.5 Coût de migration d'une tâche 110

6.4.6 Surcoût associé à l'accès aux variables partagées 110

6.4.7 Code généré 111

6.5 Modèle de programmation 112

6.5.1 Influence de la MPV 114

6.5.2 Migration de tâches 116
6.6 Résumé 117

7 Conclusion **118**

A Programmes ParSubC utilisés **120**

A.1 Versions parallèles 121
A.1.1 `abisort` 121
A.1.2 `fib` 125
A.1.3 `mm` 126
A.1.4 `mm-variante` 129
A.1.5 `poly` 130
A.1.6 `queens` 132
A.1.7 `scan` 133
A.1.8 `sum` 135
A.2 Versions séquentielles 137
A.2.1 `mm` 137
A.2.2 `scan` 139
A.2.3 `sum` 140
A.3 Versions parallèles avec migration de tâches 141
A.3.1 `sum_mt` 141
A.3.2 `scan_mt` 143
A.3.3 `square` 146

B Profils d'exécution **148**

B.1 `fib(30)` 150

B.2 `fib(35)` 150

B.3 `12-queens` 151

B.4 `13-queens` 151

B.5 `mm` 152

B.6 `mm-variante` 152

B.7 `abisort` 153

B.8 `poly` 153

B.9 `scan` 154

B.10 `sum` 154

B.11 Bonne exécution de `scan_mt` 155

B.12 Mauvaise exécution de `scan_mt` 155

Bibliographie **156**

Liste des tableaux

4.1 Structures de données et pointeurs utilisés dans le code 43

6.1 Accélérations avec le Cray T3D. 93

6.2 Accélérations avec le réseau de stations HP. 95

6.3 Performances avec la migration de tâches 99

6.4 `square` avec différents partitionnements de données 103

6.5 Mesures sur le partitionnement. 106

6.6 Facteur de dérangement des stations libres. 107

6.7 Impact du "polling" dans l'implantation avec mémoire partagée 108

6.8 Temps d'accès à la MPV. 109

6.9 Surcoût associé aux variables partagées 111

6.10 Compilateur ParSubC versus C. 112

Table des figures

1.1 Parallélisme avec Concurrent C 3

1.2 Parallélisme avec IRIS Power-C 5

1.3 Code C du programme `fib.c` 11

1.4 Code généré pour le programme `fib` 12

2.1 Exemple d'exécution parallèle avec ParSubC 17

2.2 Équivalence de `!!` en modèle "fork-join" explicite 18

2.3 Partitionnement récursif 19

2.4 Suite de Fibonacci avec ParSubC 20

2.5 Parallélisme de données avec ParSubC 22

3.1 Problème avec la migration du bloc 25

3.2 Opérations sur la pile de tâches légères en fonction de la taille et l'âge des tâches 26

3.3 Empilement d'une tache légère 31

3.4 Code généré pour la fonction `pfib`. 32

3.5 Vol de tâche 34

3.6 Points de contrôle de `pfib.c` 35

3.7 Exécution d'une tâche volée 36

3.8 Retour de vol de tâche 37

3.9 Fonction de recherche de travail (get_work) 39

4.1 Composantes et leurs interactions 45

4.2 Éléments composants un message 49

4.3 Accès partagé à une variable locale 51

4.4 Mémoire partagée virtuelle 56

4.5 Exemple d'annotation de pointeurs 58

4.6 Migration de tâche: pointeur sur une variable locale 60

4.7 Indirection des variables locales *à risque* 61

4.8 Code de migration de tâche 64

4.9 Tâche légère associée à une tâche à migrer. 66

4.10 Ordonnancement et pointeurs _hp et _tp. 68

4.11 Nouvelle source de travail pour la fonction get_work 70

5.1 Structure d'un processeur 74

5.2 Structure de variables globales 76

5.3 Vol de tâche et dépilement simultané d'une tâche (PMP) 76

5.4 Partitionnement et récursivité 77

5.5 Création de tâche par le voleur 81

5.6 Retour d'un vol 82

5.7 Envoi d'une requête de vol 83

5.8 "polling" 87

6.1 Courbes d'accélération pour queens, fib, sum et scan — Cray T3D . 92

6.2 Courbes d'accélération pour poly, mm et abisort — Cray T3D 94

6.3 Courbes d'accélération de queens, poly, sum et fib — Multi-ordinateur 96

6.4 Courbes d'accélération pour sum_mt, scan_mt et square 100

6.5 queens et fib avec granularité plus élevée sur le multi-ordinateur. . . . 105

6.6 Variante de mm pour diminuer le nombre de diffusions. 115

B.1 Profil d'exécution de fib30 sur le multi-ordinateur. 150

B.2 Profil d'exécution de fib35 sur le multi-ordinateur. 150

B.3 Profil d'exécution de 12-queens sur le multi-ordinateur. 151

B.4 Profil d'exécution de 13-queens sur le multi-ordinateur. 151

B.5 Profil d'exécution de mm sur le multi-ordinateur. 152

B.6 Profil d'exécution de mm-variante sur le multi-ordinateur. 152

B.7 Profil d'exécution de abisort sur le multi-ordinateur. 153

B.8 Profil d'exécution de poly sur le multi-ordinateur. 153

B.9 Profil d'exécution de scan sur le multi-ordinateur. 154

B.10 Profil d'exécution de sum sur le multi-ordinateur. 154

B.11 Profil d'exécution de scanmtok sur le multi-ordinateur. 155

B.12 Profil d'exécution de scanmtbad sur le multi-ordinateur. 155

Chapitre 1

Introduction

Ce mémoire porte sur la conception d'une variante parallèle du langage C, appelée ParSubC, conçue pour exprimer du parallélisme de contrôle. La variante proposée a un modèle de programmation simple mais polyvalent et s'utilise facilement avec la récursivité, ce qui facilite le partitionnement de tâches et peut offrir de bonnes performances.

Bien que le langage offre un modèle de programmation à mémoire partagée, la principale implantation a été réalisée pour un multi-ordinateur à mémoire distribuée. Ceci montre que le langage est portable et est en quelque sorte indépendant de l'architecture d'ordinateur. Cette implantation est basée sur la technique efficace de partitionnement nommée "création paresseuse de tâche".

1.1 Motivation

Les applications informatiques qui sont développées de nos jours sont de plus en plus exigeantes au niveau du temps de calcul, c'est pourquoi il peut être très intéressant de paralléliser ces applications pour en améliorer les performances.

Il existe des compilateurs parallélisants [LPT92] [Pol88] [ZC91], c'est-à-dire qui identifient et exploitent le parallélisme qui existe dans un programme écrit dans des langages séquentiels. Ces compilateurs ne peuvent détecter et exploiter que des formes très simples de parallélisme telles que dans des boucles. Pour certains programmes de telles transformations ne sont pas suffisantes pour obtenir de bons résultats. Pour avoir de meilleures performances, il est préférable de se tourner vers des langages qui permettent d'exprimer explicitement le parallélisme. Avec ces langages, c'est le programmeur qui a

la responsabilité d'exprimer le parallélisme contenu dans ses programmes. Les performances possibles avec ces langages sont potentiellement supérieures à celles obtenues avec des compilateurs parallélisants car le programmeur peut concevoir et restructurer ses programmes de façon à en exposer le parallélisme, ce qui est une tâche très difficile à réaliser pour un compilateur.

Il existe déjà plusieurs variantes de C à parallélisme explicite, mais la majorité de ces langages sont orientés vers le parallélisme de données (Cid [Nik94], Dataparallel C [HQ91], C* [RS87], HPC [DBF94]). Il est possible avec ce type de parallélisme d'améliorer les performances des problèmes qui nécessitent des calculs simples et réguliers sur des données, mais ces problèmes forment une classe plutôt restreinte. Pour les autres problèmes, le parallélisme de données n'est pas adéquat, car ils ont souvent une forme de parallélisme plus complexe et demandent ainsi un modèle de parallélisme plus souple, d'où l'attrait du parallélisme de contrôle.

Les langages à parallélisme de contrôle sont souvent pour *connaisseurs*, c'est-à-dire qu'il faut acquérir beaucoup d'expérience avant de pouvoir les utiliser pleinement. On en retrouve d'autres qui sont plus simples, mais généralement le prix à payer pour cette simplicité est un modèle de programmation faible. Il serait intéressant d'avoir un langage pour le parallélisme de contrôle qui soit simple tout en offrant un modèle de programmation souple et puissant.

1.2 Langages destinés au parallélisme de contrôle

Lorsqu'un langage parallèle destiné au parallélisme de données est conçu, partitionner le travail et avoir un degré de parallélisme suffisamment élevé pour que tous les processeurs aient du travail sont des problèmes assez simples à régler car ils sont directement reliés au partitionnement des données. Tous les processeurs exécutent le même programme de façon synchrone, mais chacun sur les données qui lui sont attribuées. Pour ce qui est du parallélisme de contrôle, les solutions possibles sont plus variées et il faut y porter une attention particulière, car la solution retenue aura une grande influence sur la qualité du langage (par exemple la simplicité, la performance et la portabilité).

1.2.1 Concurrent C

Un premier exemple de langage à parallélisme explicite pour le parallélisme de contrôle est Concurrent C [GR89]. Ce langage étend le langage C pour offrir au pro-

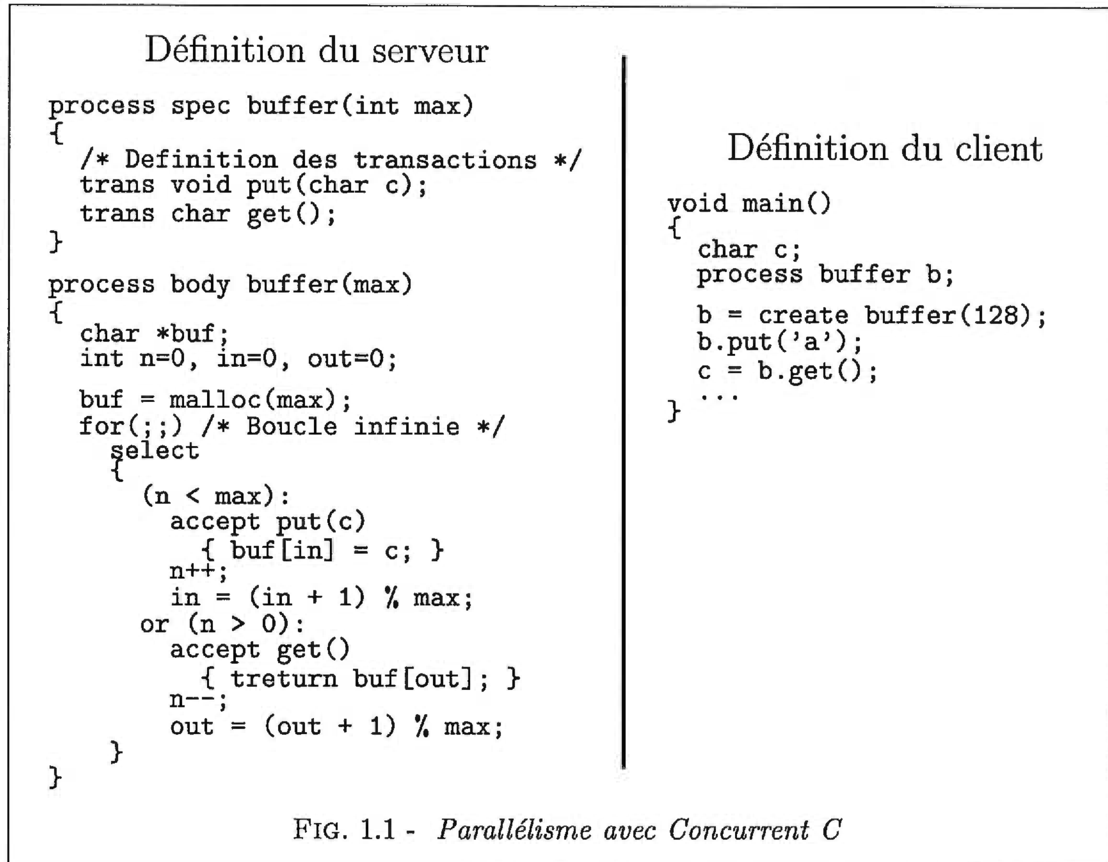

FIG. 1.1 - *Parallélisme avec Concurrent C*

grammeur la possibilité de créer dynamiquement des processus à l'intérieur de ses programmes. Dans Concurrent C, un processus est une fonction que le programmeur définit dans son programme et qu'il déclare comme étant de type `process` (voir la figure 1.1). Le parallélisme prend place lorsqu'il y a création de ces processus, car ceux-ci sont immédiatement créés et peuvent alors être exécutés en parallèle avec celui qui les a créés, soit le *parent*.

Le parent peut ensuite communiquer avec le processus créé à l'aide de *transactions*. Dans notre exemple, deux transactions sont disponibles, soit `put` et `get`. Ainsi le parent peut demander au processus d'ajouter un caractère dans le tampon (`b.put('a')`) ou d'en retirer un (`b.get()`). De son côté, le processus est défini comme pouvant accepter ces deux transactions sous des conditions spécifiques, avec `accept put()` et `accept get()`.

L'exemple de la figure 1.1 est trivial, mais on peut bien y voir la philosophie derrière Concurrent C. Ce langage permet de définir un modèle client-serveur, dans lequel on définit des serveurs ainsi que toutes les différentes actions qu'ils peuvent exécuter. En-

suite, à l'aide de transactions, un client peut faire exécuter en parallèle, par les serveurs, ces différentes actions.

Paralléliser un programme avec Concurrent C nécessite la restructuration de celui-ci pour y définir des serveurs (de type `process`) et pour ensuite y ajouter des transactions. Comme mentionné dans [AGG+], la quantité et la complexité des ajouts à C qu'on retrouve dans Concurrent C le classent plutôt comme étant un nouveau langage que comme étant une simple extension à C. Ainsi, pour le programmeur, le passage de C à Concurrent C nécessite l'apprentissage de plusieurs nouveaux concepts.

Pour illustrer le reproche que l'on fait à Concurrent C, il est intéressant de rappeler qu'il est possible, lorsqu'on veut paralléliser un programme, de le faire en utilisant que le langage C standard avec des librairies qui contiennent des fonctions permettant de démarrer des processus sur d'autres stations (par exemple la fonction `rsh`). Il est alors possible d'écrire différents programmes (serveurs) qui seront démarrés par un programme principal (client). À l'aide de fonctions de communications simples, il est aussi possible de faire des transactions. Cependant la complexité de cette méthode est évidemment trop grande pour en faire une méthode accessible à tous les programmeurs, car il y a beaucoup de détails et de connaissances à posséder avant de pouvoir faire de la programmation parallèle de cette façon. Une caractéristique importante d'un langage parallèle de haut niveau est qu'il doit faciliter la programmation parallèle en cachant le plus possible les détails de partitionnement et de communication.

Concurrent C facilite beaucoup la tâche pour ce qui est de créer des processus et de communiquer entre ces processus (transactions). Malgré cela, programmer avec ce langage nécessite encore beaucoup de travail de la part du programmeur face au parallélisme. Ainsi il doit donc penser et structurer ses programmes en fonction du parallélisme, et doit également gérer explicitement les différents processus. On peut donc reprocher à ce langage un manque de simplicité autant au niveau syntaxique que sémantique.

De plus, même avec une bonne implantation, la création de processus et la communication entre ceux-ci sont des opérations assez lentes. Ainsi, il ne sera pas possible d'obtenir de bons résultats avec des programmes parallèles dont la granularité des tâches est fine.

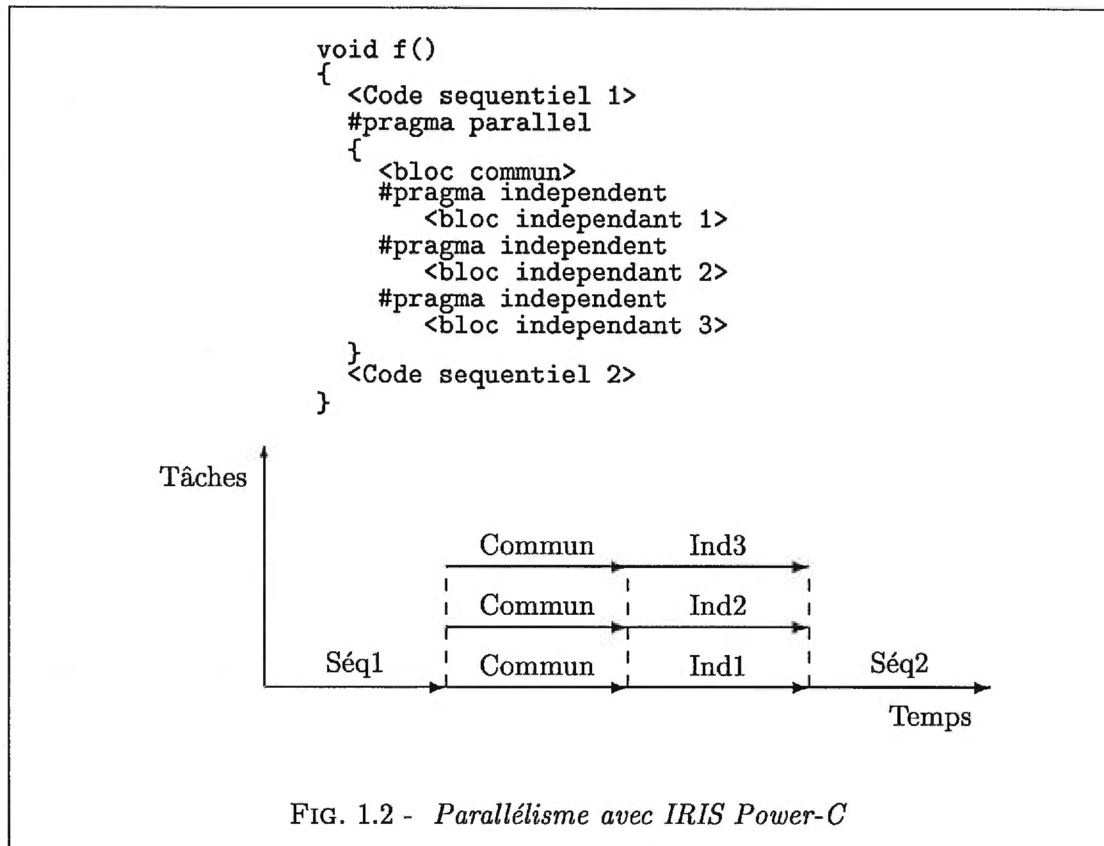

FIG. 1.2 - *Parallélisme avec IRIS Power-C*

Concert/C

Concert/C [AGG+] est un langage similaire à Concurrent C. Malgré que sa syntaxe soit moins lourde et qu'il soit plus simple que Concurrent C, Concert/C offre également un modèle de programmation parallèle basé sur le modèle client-serveur. Similairement à Concurrent C, il faut avec ce langage définir et créer des processus (serveurs) avec lesquels les clients peuvent faire des transactions. La complexité de ce langage est également sa principale faiblesse.

Concurrent C et Concert/C sont en quelque sorte des langages pour *connaisseurs*, c'est-à-dire qu'ils nécessitent beaucoup de connaissances avant de pouvoir être utilisés.

1.2.2 IRIS Power-C

IRIS Power-C [Bau92] est un bel exemple de langage simple avec lequel il est très facile d'écrire un programme parallèle. La figure 1.2 montre un programme et un exem-

ple de son exécution sur trois processeurs. Comme on peut le voir, il suffit d'annoter un bloc avec `#pragma parallel` pour que celui-ci soit exécuté en parallèle. L'annotation `#pragma independant` indique que le bloc qui suit ne doit être exécuté que par un seul processeur. Sans cette annotation, le bloc serait exécuté par tous les processeurs (comme c'est le cas avec le bloc `commun`).

Il existe plusieurs autres annotations. Entre autres, nous en retrouvons pour définir des sections critiques, pour indiquer l'utilisation des variables locales, etc. On peut même exprimer du parallélisme de données grâce à `#pragma pfor` qui permet l'exécution parallèle d'une boucle `for`.

L'idée d'utiliser la forme `#pragma` est intéressante car cela fait déjà partie du langage C, ce qui rend la syntaxe du langage Power-C compatible à cent pour cent avec celle du C. Lorsque compilées avec un compilateur C standard, les expressions `#pragma` inconnues sont simplement ignorées et la compilation continue normalement.

Il est donc facile d'écrire un programme parallèle avec IRIS Power-C, car il suffit de spécifier dans un bloc `#pragma parallel` les différentes tâches indépendantes qui peuvent être exécutées en parallèle. La syntaxe du langage est simple et permet d'exprimer différentes formes de parallélisme.

Le principal reproche qu'on peut cependant faire à ce langage est d'avoir un partitionnement statique, c'est-à-dire que le travail est découpé en un certain nombre de tâches lors de l'écriture et ce nombre ne peut changer lors de l'exécution. Le degré de parallélisme du programme est donc fixe et la seule méthode pour l'augmenter est de modifier et recompiler le programme. Pour illustrer l'inconvénient du partitionnement statique, il suffit de se demander ce qui arriverait si on exécutait le programme de la figure 1.2 sur plus de trois processeurs. La réponse est que pendant l'exécution des blocs indépendants, seulement trois processeurs travailleraient tandis que les autres n'auraient rien à faire. Il arrive que le degré de parallélisme soit limité par le programme lui-même, mais pour les programmes où ce n'est pas le cas, il serait préférable d'avoir un partitionnement dynamique qui permettrait à l'exécution d'adapter le nombre de tâches au nombre de processeurs qui participent à l'exécution.

Une façon de remédier à ce problème aurait été d'utiliser la récursivité. De cette façon, lorsqu'une fonction contient un bloc parallèle et que, dans ce bloc on retrouve un appel récursif à cette même fonction, il serait possible de faire varier le nombre de tâches disponibles pour l'exécution parallèle simplement en faisant varier le nombre d'appels récursifs. Malheureusement, peut-être pour des raisons d'implantation, le langage ne supporte pas le parallélisme récursif. Ainsi avec Power-C, les blocs parallèles des appels

récursifs sont exécutés de façon séquentielle.

Comme on vient de le voir, IRIS Power-C est un langage simple et facile à utiliser. Cependant les extensions qui s'y retrouvent sont trop spécialisées et rendent le langage inadéquat à l'usage générale.

1.3 Architectures

Parce que ParSubC est destiné au parallélisme de contrôle, le type d'architecture visé pour ce travail est de type "multiple instruction, multiple data" (MIMD). Pour permettre d'évaluer le langage, tant au niveau du modèle de programmation qu'au niveau de la performance, une première implantation a été réalisée sur un multi-ordinateur à mémoire distribuée constitué d'un réseau de stations de travail. Comme nous pourrons le voir dans le chapitre sur la performance, c'est un type d'architecture parallèle où les communications inter-processeur sont lentes, et cela diminue donc les performances du langage dans certains problèmes. Une deuxième implantation a ainsi été réalisée mais cette fois pour un ordinateur multi-processeur à mémoire partagée, qui est une architecture où les communications inter-processeur sont beaucoup plus rapides. Cette deuxième implantation nous servira de point de comparaison pour la première.

Si l'ordinateur multi-processeur offre un support pour notre langage plus performant que le multi-ordinateur, on peut alors se demander pourquoi l'avoir implanté sur un multi-ordinateur. La raison est qu'un multi-ordinateur possède deux avantages assez intéressants par rapport au multi-processeur:

- Les multi-ordinateurs à mémoire distribuée, comme un réseau de station de travail, sont beaucoup plus populaires et donc beaucoup plus disponibles comme architecture.
- Les multi-ordinateurs offrent une extensibilité intéressante au niveau du nombre de processeurs disponibles.

Il est intéressant de noter que nous avons pu vérifier le premier point énoncé ci-haut, car à deux reprises lors de la réalisation de ce projet, nous avons perdu l'accès à un ordinateur multi-processeurs avant d'avoir complètement terminé nos expériences.

1.3.1 Cray T3D et réseau de stations HP

Le multi-processeur utilisée pour ce mémoire est un Cray T3D de Cray qui contient 512 processeurs DEC[1] Alpha d'une vitesse de 150 Mhz et ayant chacun une mémoire locale de 64 Mo. Les processeurs sont pairés en noeuds qui sont interconnectés sous forme d'un toroïde à 3 dimensions. Le Cray T3D ne possède pas une mémoire unique pour les données partagées, mais grâce au réseau d'interconnexion, chaque processeur peut accéder à la mémoire de tout autre processeur. Tout ceci se fait de façon strictement matérielle et sans l'intervention des processeurs, ce qui permet de classer le Cray T3D dans la famille des ordinateurs à mémoire partagée. Cette mémoire partagée est de type NUMA[2], c'est-à-dire que le temps d'accès varie en fonction de la distance. Les temps d'accès sont de 30 cycles machine pour un accès local et entre 90 et 120 cycles pour un accès à une mémoire non-locale, dépendant de la distance entre les deux noeuds. Un accès non-local est donc entre 3 et 4 fois plus lent qu'un accès local.

Le multi-ordinateur utilisé est un réseau de stations HP-715D de Hewlett Packard ayant chacune un processeur d'une vitesse de 75 Mhz avec une mémoire de 32 Mo ayant un temps d'accès de 70 *ns* (avec un bus de 64 bits). Le réseau qui les relie est de type Ethernet avec une vitesse de transfert maximale de 8 Mbits/sec. Contrairement au réseau interne du Cray T3D où un message est acheminé à un noeud par un chemin précis, lorsqu'un message est envoyé sur le réseau des stations, celui-ci est diffusé à toutes les stations, c'est-à-dire que toutes les stations reçoivent le message. Les stations doivent constamment *écouter* sur le réseau pour vérifier si un message leur est destiné. Cependant, cela se fait grâce à du matériel spécialisé et aucune intervention du processeur n'est requise lorsqu'un message ne lui est pas acheminé. Contrairement au temps d'accès mémoire qui est de l'ordre des nano-secondes, un temps d'accès au réseau (envoi et réception d'un message) typique est plutôt de l'ordre des millisecondes. La différence entre réseau et mémoire est donc beaucoup plus importante pour notre multi-ordinateur que pour le Cray T3D.

1.4 Modèle de programmation à mémoire partagée

La programmation parallèle multi-ordinateur à mémoire distribuée amène plusieurs questions face à la gestion et l'utilisation de l'espace mémoire. La difficulté avec cette architecture est qu'au niveau matériel, la mémoire de chaque station est indépendante et

1. Digital Equipment Corporation
2. "non-uniform memory access"

il est donc impossible pour une station d'accéder directement à la mémoire d'une autre station. Il est cependant essentiel, pour pouvoir exécuter parallèlement un même programme, que les stations communiquent entre elles, sinon le partitionnement de tâches ne pourrait avoir lieu. Comme le réseau est le seul lien que possèdent les stations pour communiquer, il faudra utiliser ce support pour distribuer le travail entre les stations.

Le partitionnement est un premier élément qui nécessite la communication entre les stations, mais ce n'est pas le seul. Programmer avec un langage impératif comme le langage C consiste à changer l'*état* d'un programme pour arriver à un état final qui consiste en la solution. Ce type de langage repose fortement sur l'effet de bord provenant de l'affectation, qui permet ainsi de modifier les variables qui définissent l'état du programme [Set90]. Se pose donc le problème à savoir comment les stations pourront partager l'état du programme. Avec le parallélisme de données, on distribue les données entre les stations et chaque station est responsable d'effectuer les calculs sur ses données. Ainsi chaque station a son propre état et l'état final, qui correspond à la solution au problème, sera constitué de tous les états finaux des stations. Il faut cependant noter que c'est là une présentation très simplifiée des langages pour parallélisme de données.

Cependant, comme ParSubC est destiné au parallélisme de contrôle, cette solution n'est pas adéquate et le problème du partage de l'état du programme est toujours présent. Si nous nous limitions à ne considérer que l'aspect *fonctionnel* du langage C, comme dans le programme de la figure 1.3, l'exécution d'un programme se résumerait à l'évaluation d'une fonction et le problème du partage de l'état du programme ne se poserait donc pas. Avec la programmation fonctionnelle *pure*, résoudre un problème se fait uniquement par l'appel de fonction et, il est maintenant possible de réduire le problème de parallélisation d'un programme à celui du partitionnement des appels de fonctions entre les stations. Un exemple de langage qui illustre bien ce type de parallélisme est le langage fonctionnel parallèle Multilisp [Hal85].

En tant que langage impératif destiné au parallélisme de contrôle, ParSubC devra permettre aux stations de partager un état commun. Il reste maintenant à définir de quel moyen disposeront les stations pour partager cet état. Le choix fait par ParSubC est de fournir un *modèle de programmation à espace d'adressage unique*, c'est-à-dire qu'un programme ParSubC pourra être écrit en supposant la présence d'une mémoire unique et ainsi, l'accès aux variables est la même qu'en C. La principale motivation de ce choix est que c'est un modèle de programmation plus simple puisqu'il permet au programmeur de faire abstraction de l'architecture distribuée sur lequel les programmes s'exécuteront. Ainsi un programme ParSubC pourra être exécuté aussi bien sur un ordinateur à mémoire partagée qu'un à mémoire distribuée.

1.4.1 Mémoire partagée virtuelle

Comme il a été vu précédemment, dans un multi-ordinateur à mémoire distribuée, le seul lien direct qui existe entre processeur et mémoire est celui d'un processeur et sa propre mémoire. Il n'existe pas de lien direct entre un processeur d'une station de travail et la mémoire d'une autre station. Ainsi le seul moyen que possèdent nos stations de travail pour communiquer entre elles est par envoi de messages à travers le réseau. Pour offrir un modèle de programmation à mémoire partagée dans notre implantation de ParSubC avec mémoire distribuée, l'accès à une variable de la mémoire partagée pourra nécessiter l'envoi de messages. Certaines variables auront un emplacement unique et donc, l'accès à une de ces variables nécessitera l'envoi d'un message au propriétaire de la variable (sauf évidemment si l'accès est fait par le propriétaire). D'autres variables seront logées dans la mémoire de chaque station et ainsi une référence à ces variables ne nécessitera pas d'accès au réseau (puisque chaque station en possède une copie). Cependant, la mise-à-jour de ces variables impliquera la diffusion d'un message à toutes les stations pour garder la cohérence entre toutes les copies. Plus de détails seront donnés sur la mémoire partagée virtuelle dans le chapitre 4.

Programmer en ParSubC sur notre multi-ordinateur à mémoire distribuée revient en quelque sorte à programmer sur un ordinateur à mémoire partagée de type NUMA avec un coût accès à une mémoire non-locale extrêmement élevé. Comme nous le verrons lors de l'évalution des implantations, le temps d'accès à la mémoire non-locale se compte en nano-secondes pour le Cray T3D contrairement à des milli-secondes avec la mémoire partagée virtuelle de notre multi-ordinateur. Nous verrons également l'effet des mémoires partagées de type NUMA sur l'exécution des programmes parallèles.

1.5 Code généré

Lorsque les compilateurs ParSubC compilent un programme, ils génèrent du code C qui sera compilé et lié à des librairies nécessaires à l'exécution parallèle. Le temps d'exécution du programme est plus élevé que si du code machine avait été généré, mais le code C donne au compilateur une portabilité très intéressante. Puisque le principal intérêt de ce travail est d'évaluer le modèle de programmation du langage et les accélérations qu'il est possible d'obtenir avec ce langage, nous trouvons qu'il est justifiable de payer ce coût de ralentissement pour la portabilité offerte. En plus, le code C offre une simplicité de programmation qui a permis d'implanter plus rapidement le compilateur que si du code machine avait été généré.

```
int fib(int i)
{
  if (i < 2) return i;
  else return fib(i-1) + fib(i-2);
}

void main()
{
  fib(40);
}
```

FIG. 1.3 - *Code C du programme* `fib.c`

La transformation de ParSubC à C consiste à décomposer les énoncés que l'on retrouve dans le programme initial en opérations de base, de façon à avoir un meilleur contrôle sur le programme et ainsi pouvoir y introduire les opérations nécessaires au parallélisme. Le programme C généré est similaire à du langage machine, c'est-à-dire que seulement des opérations simples telles l'affectation, les opérations arithmétiques, des `goto`, etc., sont utilisées. Ainsi toutes les formes plus complexes telles les boucles, les appels de fonctions, etc. qu'on retrouve dans le programme ParSubC initial, sont décomposées en ces opérations de base. La figure 1.3 montre un petit programme ParSubC (sans parallélisme) et la figure 1.4 montre sa décomposition en langage C. Notre but n'est pas d'expliquer la transformation en détail, mais simplement de faire ressortir les points importants.

Comme on peut le voir dans la figure, le programme ParSubC est converti en une seule fonction, soit la fonction `_execute`. Dans cette fonction, on retrouve principalement comme structure de contrôle, des étiquettes et un énoncé `switch`. Les étiquettes permettent de décomposer les structures de contrôle complexes du programme ParSubC. Ainsi une boucle `for`, par exemple, sera réécrite en utilisant des `goto` et des étiquettes. La décomposition des énoncés se fait donc de façon similaire à du code machine. De plus, les variables locales sont placées dans une pile qui est gérée explicitement par le code. La variable `_fp` correspond au pointeur de pile ("frame pointer") et tout accès aux variables dans la pile se fait au travers de ce pointeur. Ainsi, un accès à la variable locale `i` de la fonction `fib` de notre exemple est remplacée par `*(int*)(_fp+0)`. Ici, le déplacement est nul car la variable est au bas du bloc d'activation de la fonction `fib`, mais si cette fonction avait une deuxième variable locale, un accès à cette variable correspondrait à `*(int*)(_fp+1)`.

L'énoncé `switch` et les `case` répartis à travers le code permettent de traduire les retours de fonction (et comme on le verra plus loin, le partitionnement du travail). À

```
void _execute()
{
  _INIT_EXECUTE(); /* Prepare l'execution de la tache */
_return:
  /* Extraction de l'adresse de retour. */
  _pc = *(int*)(_fp-1);
_jump:
  switch (_pc)
  {
    _CASE_0(); /* Macro pour le cas 0 qui correspond a la fin
                  d'execution d'une tache. */
__fib: case 1: /* Fonction fib */
    / On verifie si (i<2) */
    if ((*(int*)(_fp+0))<(2))
    {
      *(int*)(_fp-(1+1)) = *(int*)(_fp+0);
      goto _return;
    }
    else
    {
      /* Preparation de l'appel de fonction fib(i-1) */
      *(int*)(_fp+3) = (*(int*)(_fp+0))-(1);
      _fp += 3;
      *(int*)(_fp-1) = 2;
      goto __fib; /* Appel de fonction fib(i-1) */
case 2:;
      /* Preparation de l'appel de fonction fib(i-2) */
      _fp -= 3;
      *(int*)(_fp+4) = (*(int*)(_fp+0))-(2);
      _fp += 4;
      *(int*)(_fp-1) = 3;
      goto __fib; /* Appel de fonction fib(i-2) */
case 3:;
      /* Retour du resultat */
      _fp -= 4;
      *(int*)(_fp-(1+1)) = (*(int*)(_fp+1))+(*(int*)(_fp+2));
      goto _return;
    }
    goto _return;
__main: case 4: /* Fonction main */
    /* Preparation de l'appel de fonction fib(40) */
    *(int*)(_fp+2) = 40;
    _fp += 2;
    *(int*)(_fp-1) = 5;
    goto __fib; /* Appel de fonction fib(40) */
case 5:;
    _fp -= 2;
    goto _return; /* Retour de la fonction main */
  }
}
```

FIG. 1.4 - *Code généré pour le programme* `fib`

chaque point de retour d'une fonction du programme ParSubC correspond un `case` du `switch`. Pour retourner d'une fonction (voir l'étiquette `_return`), il suffit d'extraire l'*adresse* de retour du bloc d'activation et de brancher au `case` correspondant à l'aide du `switch`.

Il y a donc une forte ressemblance entre le code C généré et du code machine. L'avantage du code C est sa portabilité. En contre-partie, comme nous pourrons le voir, le code C généré est moins efficace que du code machine.

Chapitre 2

Concepts de programmation parallèle

Le but premier de ce travail est de concevoir et réaliser un langage à parallélisme explicite pour le parallélisme de contrôle. L'analyse de certains langages a permis de voir certaines faiblesses de ce type de langage, et a permis d'identifier des aspects ou concepts que nous ne voudrions pas voir dans notre langage. Maintenant, la prochaine étape pour permettre de définir notre langage est d'identifier ce qui est nécessaire ou désirable de voir dans notre langage.

D'après les critiques faites aux langages dans le chapitre précédent, on peut énoncer deux objectifs visés par notre langage, soit la simplicité, mais tout en ayant une certaine souplesse et puissance. Cependant, ce sont des termes plus ou moins précis, et ce que cela implique vraiment pour un langage, n'est pas très clair. Il est donc souhaitable de définir quelques concepts et caractéristiques propres aux langages destinés au parallélisme de contrôle, afin de mieux définir la simplicité et la puissance d'un langage.

2.1 Caractéristiques d'un langage parallèle

2.1.1 Disponibilité du langage

Parmi les différentes caractéristiques existantes ayant trait à un langage à parallélisme explicite, on peut en mentionner trois qui selon [Hwa93] sont importants pour rendre le langage disponible, c'est-à-dire qu'il soit facilement accessible aux programmeurs.

Compatibilité

Moins un langage contient de concepts et d'éléments nouveaux, plus le langage sera facilement accessible puisqu'il requiert moins d'apprentissage avant de l'apprivoiser. Ainsi, pour un nouveau langage parallèle, la compatibilité avec un langage séquentiel existant et populaire est un facteur qui peut grandement contribuer à sa disponibilité. De plus, la compatibilité est un élément très favorable à la simplicité d'un langage.

Portabilité

Le langage ne doit pas être dépendant d'une architecture spécifique. Il doit pouvoir être facilement porté d'une architecture parallèle à une autre. Il doit idéalement pouvoir être supporté par une architecture avec mémoire partagée aussi bien que par une architecture avec mémoire distribuée. Être supporté par un grand ensemble d'architectures parallèles favorise la disponibilité du langage.

Extensibilité

Il est préférable pour notre langage que le nombre de processeurs avec lesquels un programme est exécuté soit variable et qu'il puisse s'adapter au nombre de processeurs disponibles à l'exécution. Un programme pourra ainsi être utilisé sans modifications sur des architectures dont le nombre de processeurs est plus petit ou plus grand que lors du développement, et à l'exécution, il pourra quand même tirer profit du nombre de processeurs disponibles. Cette caractéristique augmente en quelque sorte la portabilité des programmes.

2.1.2 Gestion des tâches

Contrairement au parallélisme de données où tous les processeurs exécutent simultanément le même programme et donc ont le même flot de contrôle, le parallélisme de contrôle divise plutôt le travail en plusieurs tâches distinctes. Le travail est *partitionné* en plusieurs tâches que les processeurs peuvent exécuter en parallèle. On distingue deux types de *partitionnement*, soit le partitionnement statique et le partitionnement dynamique.

Avec le partitionnement statique, le programme est décomposé lors de son écriture en un nombre de tâches précis et fixe. À l'exécution, le nombre de tâches ne change pas.

Contrairement au partitionnement statique, le partitionnement dynamique tire profit de l'information supplémentaire qui est disponible à l'exécution. À l'écriture, le programme contient en quelque sorte des indications quant à la façon dont peut être décomposé le problème, mais le partitionnement ne se fait vraiment qu'à l'exécution. L'avantage de cette méthode est qu'à l'exécution, la taille des tâches ainsi que la disponibilité des processeurs sont connues, ainsi le programme peut être partitionné pour tirer profit au maximum des conditions d'exécution. De plus, contrairement au partitionnement statique, le degré de parallélisme est variable et offre donc une meilleur extensibilité.

Le partitionnement dynamique est souvent accompagné d'un *balancement de charge automatique*. Par *balancement de charge* on entend la répartition du travail entre les processeurs. Lorsqu'un balancement de charge est dit automatique, c'est que la répartition du travail est fait à l'exécution et de façon à ce que les processeurs soient le plus souvent occupés. Dès qu'un processeur n'a plus de travail, la charge est automatiquement répartie pour lui en fournir (si possible).

Le partitionnement dynamique et le balancement de charge automatique utilisent donc l'information disponible à l'exécution sur les tâches et les processeurs, de façon à mieux décomposer et distribuer le travail. Cette méthode permet une utilisation efficace des processeurs, ce qui permet d'obtenir de meilleurs performances. Par contre, il peut y avoir un surcoût, parfois considérable, pour effectuer le partitionnement et le balancement de charge à l'exécution.

2.1.3 Modèle de programmation

Comme il a été vu dans le chapitre précédent, le modèle de programmation offert au niveau de la mémoire peut influencer la qualité du modèle de programmation. Ainsi, il

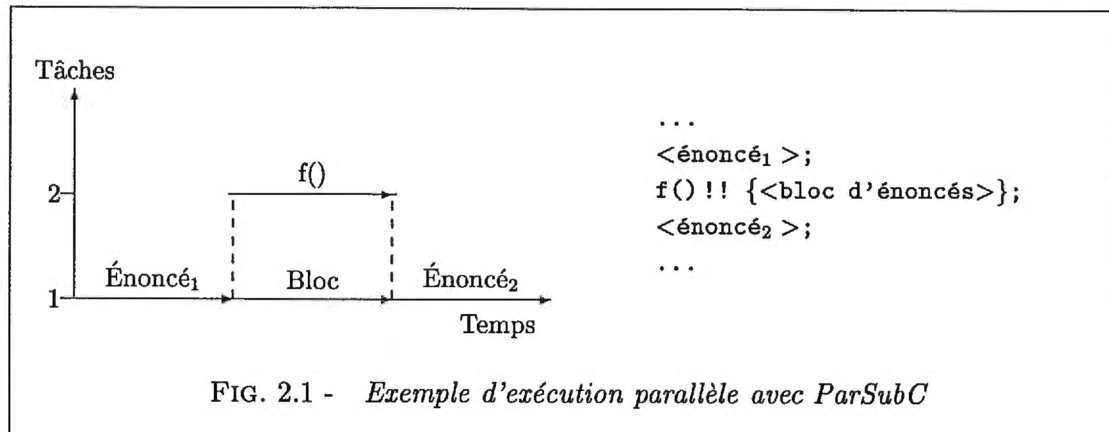

FIG. 2.1 - *Exemple d'exécution parallèle avec ParSubC*

est préférable d'offrir un modèle de programmation à mémoire partagée. De cette façon, le langage est plus simple à utiliser et cela le rend plus portable puisqu'il conserve le même modèle de programmation, qu'il soit implanté sur un multi-ordinateur à mémoire distribuée ou sur un ordinateur multi-processeur à mémoire partagée.

La granularité est un autre critère important lors de la conception d'un langage parallèle. Avec des granularités grossières et moyennes, il est facile d'obtenir de bons résultats avec un langage parallèle, car le coût associé au parallélisme a moins d'impact qu'avec une granularité fine. Mais pour être encore plus puissant un modèle de programmation parallèle doit aussi pouvoir exprimer une granularité fine. De cette façon, une plus grande classe de problèmes pourront être parallélisés efficacement. Il est à noter que la finesse de la granularité dépend également grandement de la qualité de l'implantation puisque c'est celle-ci qui gère les tâches. Une bonne implantation gardera le coût de la gestion des tâches le plus bas possible et permettra ainsi d'exprimer une granularité plus fine.

2.2 Modèle de programmation de ParSubC

Regardons maintenant le langage ainsi que le modèle de programmation qu'offre ParSubC. La variante à C proposée par ParSubC ajoute l'opérateur `!!` qui prend comme opérandes, un appel de fonction et un bloc d'énoncés. De façon informelle, sa syntaxe est la suivante:

$$fonction(expr_1, expr_2, ...) \texttt{ !! } \{enonce_1;\ enonce_2;\ ...\ \}$$

Il est à noter que ceci est une expression et qu'il est donc possible d'assigner le résultat à une variable, comme par exemple `x = f() !! {...};`.

```
...
<énoncé1>;
fork( f );
{ <bloc d'énoncés>};
join();
<énoncé2>;
...
```

FIG. 2.2 - *Équivalence de* !! *en modèle "fork-join" explicite*

L'utilisation de !! permet d'indiquer que la fonction et le bloc d'énoncés peuvent être exécutés en parallèle. Un *rendez-vous* implicite se trouve à la fin de l'exécution des deux opérandes, c'est-à-dire que l'affectation du résultat de l'appel de fonction parallèle ne peut s'effectuer que lorsque les deux opérandes ont été exécutées.

2.2.1 Modèle "fork-join"

Le modèle de programmation provenant de l'opérateur !! est de style "fork-join" [GM88], c'est-à-dire qu'à un certain point du programme, le fil d'exécution se divise ("fork") en deux pour revenir ("join") à un seul fil d'exécution plus tard. Comme on peut le voir dans la figure 2.1, la séparation permet de fournir du travail à un autre processeur.

Une particularité de notre modèle face au modèle "fork-join" est que l'opérateur !! indique à la fois le point de fourche et le point de jonction. La figure 2.2 montre la fonction de la figure précédente réécrite selon un modèle "fork-join" explicite. Malgré que l'approche de ParSubC soit moins souple que le modèle "fork-join" explicite, il en résulte un modèle de programmation beaucoup plus structuré ainsi qu'une implantation et une syntaxe plus simple.

2.2.2 Appel de fonction de style "Remote procedure call"

Étant donné la nature de la première opérande de l'opérateur !!, qui est un appel de fonction, notre modèle de programmation s'apparente au style "remote procedure call" (RPC) [SPG92]. Lorsqu'une fonction est appelée de cette façon, elle est exécutée par un autre processeur. Dans le cas de l'opérateur !!, le processeur fait exécuter la fonction par un autre processeur, et en parallèle, exécute le bloc d'énoncés. Tout cela se fait de

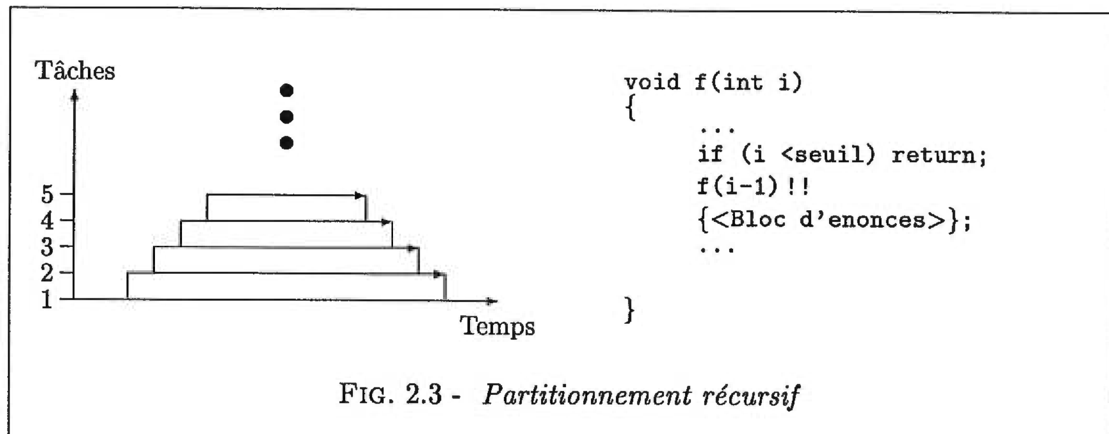

FIG. 2.3 - *Partitionnement récursif*

manière transparente.

Il y a cependant une distinction à faire entre ParSubC et le style RPC. ParSubC utilise la "création paresseuse de tâche" (CPT) comme méthode de partitionnement, et avec cette méthode, le processeur qui rencontre l'opérateur `!!` ne fait exécuter la fonction par un autre processeur que si celui-ci est libre et qu'il lui a demandé. En d'autre mot, ce sont les processeurs libres qui ont la responsabilité de se trouver du travail, et non pas celui qui rencontre `!!`. L'avantage de la CPT est de fournir un balancement de charge automatique.

2.2.3 La récursivité

À la rencontre de l'opérateur `!!`, une autre tâche devient disponible pour l'exécution parallèle. Pour exploiter plus de deux processeurs, il faut encore plus de tâches disponibles. Une première façon de le faire est d'imbriquer un `!!` dans un autre. De cette façon, à chaque niveau d'imbrication supplémentaire, une nouvelle tâche est disponible. Voici un exemple d'imbrication qui rend deux tâches disponibles:

```
f() !! { g() !! {...};};
```

Cependant, ce type de partitionnement est statique, et par conséquent, le degré de parallélisme est fixe dans le programme. Ainsi, si l'on veut modifier le nombre de tâches disponibles à l'exécution, il faut réécrire le programme en conséquence. Cette méthode de partitionnement statique n'offre donc pas une bonne extensibilité.

Pour exprimer un degré de parallélisme plus élevé, il faut utiliser la récursivité. Comme la première opérande est une fonction, il est possible de rappeler récursivement la fonction contenant `!!`, et donc à chaque tour de récursion, une nouvelle tâche est dis-

```
int pfib(int i)
{
  int t1, t2;

  if (i < 2) return i;
  t1 = pfib(i-1) !! { t2 = pfib(i-2); };
  return t1 + t2;
}
```

FIG. 2.4 - *Suite de Fibonacci avec ParSubC*

ponible. Cette méthode permet un partitionnement dynamique et donc une extensibilité beaucoup plus intéressante que l'imbrication de `!!`. Il est ainsi plus facile d'exprimer de hauts degrés de parallélisme.

La figure 2.3 montre un exemple d'exécution d'un programme utilisant le partitionnement récursif. Dans ce programme, la fonction `f` se rappelle récursivement, rendant ainsi une nouvelle tâche disponible à chaque appel récursif. Dans cet exemple, le paramètre `i` sert de condition d'arrêt comme on en retrouve normalement avec la récursivité.

Le modèle de programmation parallèle offert dans ParSubC est semblable à celui du langage IRIS Power-C avec la primitive `#pragma parallel`. Cependant IRIS Power-C ne permet pas d'exploiter le parallélisme dans les fonctions récursives. Pour être plus précis, lorsqu'il rencontre la primitive `#pragma parallel` pour une deuxième fois à cause de la récursivité, ce deuxième appel sera exécuté de façon séquentielle. Un avantage que possède clairement ParSubC par rapport à IRIS Power-C est d'exploiter le partitionnement récursif.

2.2.4 Diviser-pour-régner

Dans la section précédente, la récursivité n'a été utilisée qu'avec une seule des deux opérandes. Cependant, il n'y rien qui empêche d'avoir des appels récursifs avec les deux opérandes. Un problème qui illustre bien ce type de partitionnement est la suite de Fibonacci. L'algorithme utilisé est celui doublement récursif. Ce n'est pas l'algorithme le plus efficace mais il illustre bien la double récursivité.

Comme on peut bien le voir dans la figure 2.4, à chaque appel de la fonction `pfib` il en résulte deux autres récursifs. Par conséquent, à chaque appel de `pfib`, une autre tâche sera disponible pour être exécutée en parallèle (à moins que la condition d'arrêt n'ait été remplie). C'est donc dire qu'en un temps $\in \Omega(logn)$, il y aura n appels récursifs et donc n tâches disponibles. Ce type de partitionnement, qui est appelé *diviser-pour-régner*,

permet un partitionnement de tâche très efficace. Une discussion plus approfondie de ce type de parallélisme se trouve dans [Fee93]. Comme l'indique son nom, ce type de parallélisme permet d'exprimer des algorithmes de type diviser-pour-régner, et cela de façon très efficace.

2.3 Discussion sur ParSubC

Nous allons maintenant situer le langage ParSubC par rapport aux concepts et caractéristiques mentionnés précédemment. De cette façon, nous aurons une appréciation qualitative de ce que peut offrir ce langage.

2.3.1 Simplicité

L'extension proposée dans ParSubC est très modeste au point de vue syntaxique, ce qui facilite l'apprentissage et l'utilisation du langage. Mise à part la spécification de ce qui peut être exécuté en parallèle, il n'y a aucun détail lié au parallélisme, tels le partitionnement, le balancement de charge, la création de tâche, ... Cela fait donc de ParSubC un langage simple et rend le passage du C à ParSubC accessible à tous les programmeurs C, sans nécessiter une grande connaissance du parallélisme.

La méthode de partitionnement que l'on retrouve avec le style de parallélisme présent dans ParSubC est dynamique puisque le nombre d'appels récursifs, et donc le nombre de tâches disponibles, peut n'être connu qu'à l'exécution. Le langage est ainsi facilement extensible.

Le langage ParSubC répond ainsi à tous les critères de disponibilité mentionnés précédemment (et définis dans [Hwa93]). Le langage est *compatible* avec un langage existant bien établie. Il est indépendant de l'architecture et par conséquent *portable* et finalement, le langage est facilement *extensible* (au point de vue du nombre de processeurs).

2.3.2 Souplesse

Une caractéristique importante de ParSubC est que la classe de problèmes adressée par l'opérateur !! est vaste. C'est un langage qui permet de paralléliser beaucoup de problèmes. Dans cette classe, on retrouve tous les problèmes naturellement récursifs. Par exemple, les algorithmes travaillant sur des listes, des arbres et des graphes sont

```
void f(int debut, int fin)
{
  if (fin-debut < seuil)
  {
    /* On effectue le calcul sur cette section du tableau. */
    ...
    return;
  }
  else
  {
    int demi;
    demi = debut + (fin-debut)/2;
    f(debut, demi) !! { f(demi+1, fin); };
  }
}
```

FIG. 2.5 - *Parallélisme de données avec ParSubC*

souvent récursifs.

De plus, il est souvent possible de modifier un algorithme de façon à en faire sortir la récursivité. Par exemple, un problème typique de parallélisme de données qui consiste à faire un calcul sur chacun des éléments d'un tableau peut être facilement écrit de façon récursive. Il s'agit simplement d'appeler la fonction qui fait le calcul récursivement sur les différentes parties du tableau (voir la figure 2.5). La plupart des langages pour parallélisme de données pourraient exprimer ce problème plus simplement, mais cela montre malgré tout la souplesse de ParSubC puisqu'il peut exprimer plusieurs formes de parallélisme.

2.3.3 Puissance

Le modèle de programmation de ParSubC étant basé sur la récursivité, cela offre une méthode de partitionnement très simple mais qui peut être très efficace surtout lorsqu'utilisé sous forme diviser-pour-régner. De plus, cette méthode de partitionnement est dynamique et comme nous le verrons plus tard, elle sera implantée de manière à offrir un balancement de charge automatique. Ce type de partitionnement fait également de ParSubC un langage très facilement extensible. La simplicité et l'efficacité du partitionnement ainsi que le fait qu'il soit dynamique avec un balancement de charge automatique font de ParSubC un langage que nous pouvons qualifier de *puissant*.

2.3.4 Résumé

Le langage ParSubC offre un modèle de programmation très simple, ce qui le rend accessible à tous les programmeurs C. Grâce à son modèle de programmation basée sur la récursivité, cela en fait un modèle général et permet donc de paralléliser un grand nombre de problèmes et cela de façon efficace. On peut décrire ParSubC comme étant un langage simple, souple et puissant.

Chapitre 3

La création paresseuse de tâche

Le mécanisme de partitionnement du travail peut influencer considérablement la performance des programmes. Meilleur est le mécanisme de partitionnement, plus fine sera la granularité de tâche qu'il sera possible d'exploiter avec ce langage, ce qui permettra de paralléliser efficacement un plus grand nombre de programmes.

Pour implanter le compilateur ParSubC, la *création paresseuse de tâche* (CPT) est utilisée comme méthode de partitionnement. L'idée principale de la CPT est de retarder le plus possible la création d'une tâche, c'est-à-dire tant qu'aucun processeur n'en a besoin, dans le but de pouvoir, dans certains cas, l'éviter complètement.

3.1 Principes de la CPT

Lorsque l'exécution d'un programme ParSubC débute, tous les processeurs, sauf un, sont *libres*, c'est-à-dire sans travail. La seule tâche disponible à ce moment est celle qui exécute la fonction `main()`, et il y a donc initialement du travail que pour un seul processeur. C'est à la rencontre de l'opérateur `!!` qu'il y a plus d'une tâche disponible et que l'exécution parallèle du programme peut commencer.

3.1.1 Vol de tâche

Avec la CPT, le partitionnement est initié par les processeurs libres. Un processeur libre demande tour à tour aux autres processeurs s'ils ont du travail disponible, et ce, tant qu'il n'en aura pas trouvé. Par chercher du travail, on entend chercher une tâche à

```
int f(int a, int b)
{
  int i, j, *p;
  ...
  p = &a;
  f(2,3) !! { ... *p = 10; ... };
  ...
}
```

FIG. 3.1 - *Problème avec la migration du bloc*

exécuter. À chaque fois qu'un processeur rencontre l'opérateur !!, dans la mémoire de ce processeur est sauvegardée une indication qu'une tâche correspondant à l'appel de fonction est disponible. Ce sont ces tâches que les processeurs libres essaieront d'obtenir. On dit alors que les processeurs cherchent à *voler* une tâche.

L'opérateur !! indique que la fonction et le bloc d'énoncés peuvent être exécutés en parallèle, ainsi, il aurait été tout aussi valide de créer une tâche correspondant au bloc d'énoncés plutôt qu'à la fonction. Cependant, permettre le *vol* de la fonction est de beaucoup préférable car cela simplifie l'implantation en plus de la rendre plus efficace. Voler un bloc d'énoncés demande en général plus d'informations à communiquer au processeur qui veut voler la tâche (le *voleur*). Dans le pire cas, toutes les variables locales ainsi que tous les paramètres de la fonction courante doivent être envoyés pour que l'exécution du bloc puisse être faite correctement (dans l'exemple de la figure 3.1, il faudrait envoyer les variables `a` et `b` ainsi que `i`, `j` et `p`). Grâce à une analyse, le compilateur pourrait découvrir l'ensemble minimal de variables à communiquer, mais ceci rendrait l'implantation plus complexe. Avec le vol de la fonction, seulement les paramètres de la fonction doivent être communiqués au voleur. Ainsi, dans notre exemple, il suffirait d'envoyer les valeurs 2 et 3.

Cependant, l'argument principal en faveur du vol de la fonction vient des problèmes rencontrés avec les pointeurs lors du changement d'adresse des variables locales qui doit être effectué avec le vol de bloc. Si un pointeur, local ou global, pointe sur une variable locale à la fonction courante (le pointeur `p` et la variable `a` de la figure 3.1 par exemple), lorsque les variables locales seront migrées sur un autre processeur, le pointeur ne pointera plus sur la variable puisque celle-ci aura également migré et par conséquent, aura changé d'adresse. Il existe des solutions à ce problème, mais tout cela peut être évité en permettant le vol de la fonction plutôt que celui du bloc.

On retrouve ce même problème de pointeur qui pointe sur une variable locale avec la migration de tâche et ce problème sera abordé plus en profondeur dans le chapitre 4.

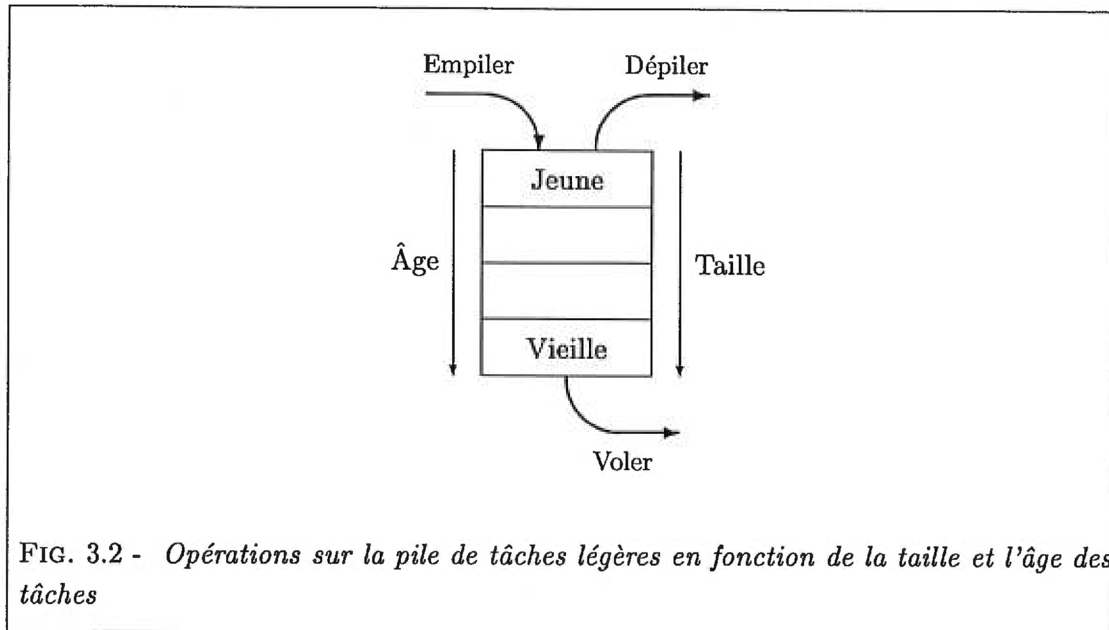

FIG. 3.2 - *Opérations sur la pile de tâches légères en fonction de la taille et l'âge des tâches*

3.1.2 Création de tâche

Lorsqu'un processeur sauvegarde une tâche, à la rencontre d'un !!, il n'y a pas immédiatement une création de tâche. L'information sauvegardée contient uniquement le minimum d'information nécessaire pour créer une tâche exécutable. On appelera *tâche légère*, les informations sauvegardées à la rencontre d'un !! et *tâche lourde*, la tâche exécutable créée à partir de la tâche légère. L'avantage que possèdent les tâches légères sur les tâches lourdes est qu'elles sont beaucoup moins coûteuses à manipuler.

C'est lorsqu'une tâche légère est volée qu'une tâche lourde doit être créée. L'avantage d'avoir deux représentations des tâches est que lorsqu'une tâche légère n'est pas volée, la tâche lourde correspondante n'est jamais créée et seulement un minimum d'information a été sauvegardé par l'exécution du !!. En retardant la création de tâche, la CPT évite ainsi de créer des tâches lourdes qui ne seront pas exécutées.

Lorsqu'un processeur termine d'exécuter un bloc d'énoncés associé à !! et que la tâche légère correspondante n'a pas été volée, il exécute localement la fonction comme un appel de fonction normal.

3.1.3 Ordonnancement des tâches

Le partitionnement récursif disponible dans ParSubC permet à un processeur d'avoir

plusieurs tâches disponibles en même temps. La CPT sauvegarde les tâches légères sur une pile locale au processeur de sorte que les nouvelles tâches soient empilées sur le dessus et que le vol de tâche s'effectue par le dessous de la pile. Comme le vol de tâche se fait par le dessous, la structure se comporte également comme une queue. La figure 3.2 présente les opérations qui peuvent être faites sur notre *pile* de tâches légères. Ces trois opérations peuvent être décrites comme suit:

- Empiler – Lorsqu'un processeur rencontre l'opérateur !!, il empile alors une tâche légère correspondant à l'appel de fonction.
- Dépiler – La tâche légère au-dessus de la pile est dépilée lorsque le processeur a terminé d'exécuter un bloc d'énoncés correspondant à l'opérateur !!. Si la tâche au-dessus de la pile n'a pas été volée, il la dépile et exécute la fonction correspondante. Si la tâche a été volée, le processeur doit attendre la fin de cette tâche avant de poursuivre l'exécution du code qui suit le !!.
- Voler – Lorsqu'un processeur reçoit une requête de vol, si la tâche du dessous de la pile est disponible, il l'enlève et l'envoie au voleur.

Donner la tâche du dessous de la pile au voleur est avantageux, car en général, les tâches plus vieilles, celles au bas de la pile, sont plus longues à exécuter que les jeunes. Avec la plupart des algorithmes récursifs, plus le niveau de récursivité est profond, plus le travail qu'il faut faire est petit. Par exemple, pour les algorithmes *diviser-pour-régner* se divisant en arbre balancé, le travail contenu dans une tâche provenant de l'opérateur !! sera une fraction du travail de la *tâche courante*. Par *tâche courante*, nous entendons celle qui a rencontré le dernier énoncé contenant l'opérateur !!. Une discussion plus approfondie sur la taille des tâches provenant d'algorithmes diviser-pour-régner se trouve dans [Fee93].

Ainsi, de façon générale, le processeur qui possède les tâches aura tendance à exécuter les plus petites et les voleurs exécuteront les plus grosses. Le vol par le dessous de la pile donnera donc lieu à moins de vol de tâches que le vol du dessus et par conséquent, lors de l'exécution d'un programme, le temps associé au vol de tâche sera moins élevé.

3.1.4 Suspension de tâche

Lorsqu'un processeur a terminé d'exécuter le bloc d'énoncés de l'opérateur !!, il vérifie si la tâche légère du dessus a été volée. Cette tâche correspond à l'appel de

fonction de l'opérateur !!. Si la tâche est toujours disponible, il y a une exécution normale de la fonction, comme si l'opérateur !! n'avait pas été utilisé, pour ensuite continuer d'exécuter la tâche courante. C'est ici qu'est la force de la CPT puisqu'en repoussant la création de tâche, celle-ci a pu être évitée, pour donner lieu à la place, à un simple appel de fonction.

Il est possible cependant que pendant l'exécution du bloc d'énoncés, la tâche du dessus ait été volée. Si l'exécution de la tâche volée est terminée, alors la tâche courante peut immédiatement poursuivre le calcul qui suit le !!. Mais si ce n'est pas le cas, elle doit alors être suspendue jusqu'à ce que la tâche volée soit terminée. Le processeur est donc libre et doit se trouver du travail. Pour avoir une exécution efficace des programmes ParSubC, il est très avantageux de réduire le nombre de suspensions de tâche, car cela revient à réduire le temps où les processeurs sont libres, à la recherche de travail.

Voilà donc un deuxième avantage à effectuer le vol en dessous de la pile. De cette façon, on réduit le nombre de suspensions de la tâche courante. Si le vol du dessus était utilisé à chaque vol de tâche, il y aura un risque de suspension de tâche, car celle-ci doit être terminée avant l'exécution du bloc correspondant, sinon il y aura suspension. Avec le vol par dessous, s'il y a n tâches disponibles dans la pile, il peut y avoir $n-1$ vols de tâche sans qu'il y ait de danger de suspension. C'est donc une façon de réduire le nombre de suspensions de tâche.

3.2 Protocoles d'accès aux piles

Avant d'entrer dans les détails de l'implantation de la CPT, nous allons discuter un peu de la méthode qu'utiliseront les processeurs pour accéder aux piles de tâches afin de s'échanger les tâches entre eux. Il existe principalement deux protocoles d'accès aux structures de données partagées qui peuvent être utilisés pour la CPT, soit le *protocole à mémoire partagée* et le *protocole à envoi de messages* [Fee93].

3.2.1 Protocole à mémoire partagée

La principale caractéristique du protocole à mémoire partagée est que le voleur manipule directement la pile de sa victime lors d'un vol de tâche. Ainsi, avec ce protocole, la responsabilité du vol de tâche repose entièrement entre les mains du voleur. Un avantage important qu'offre ce protocole est de ne pas déranger la victime lors d'un vol de tâche. Cependant, comme les piles contenant les tâches sont accessibles par tous, il

pourrait arriver que deux processeurs tentent d'accéder à la même tâche simultanément, d'où la nécessité de séquentialiser les accès faits à une pile. Cette séquentialisation a pour effet d'augmenter le temps d'accès aux piles de tâches.

De plus, le protocole à mémoire partagée impose certaines restrictions face à la mise en cache des données partagées, sinon il y aura un risque d'avoir plusieurs valeurs pour une même donnée partagée. Par exemple, si deux processeurs ont une même variable dans leur cache et écrivent dans cette variable, l'utilisation de caches cohérentes est donc requise pour mettre en cache les données partagées.

3.2.2 Protocole à envoi de messages

Avec ce protocole, un processeur ne peut accéder directement à la pile de tâches d'un autre processeur. Ainsi lors d'un vol de tâche, le voleur doit informer la victime de son désir de lui voler une tâche en lui envoyant un message. Par la suite, la victime retournera une tâche provenant de sa pile au voleur (si elle en possède). De cette façon, il n'y pas de danger d'accès simultané aux piles de tâches, et donc aucune précaution particulière ne doit être prise pour les éviter. Il y a un danger de recevoir plus d'un message de vol de tâche à la fois, mais le système de traitement des messages s'occupe de séquentialiser leur traitement. L'accès aux piles de tâche est ainsi plus efficace qu'avec le protocole précédent, mais le vol de tâche nécessite maintenant l'envoi de messages entre les processeurs ainsi que le dérangement de la victime pour que celle-ci puisse répondre à la requête de vol, et ce coût survient aussi bien lorsqu'il y a une tâche à voler que lorsqu'il n'y en a pas.

3.2.3 Protocole présenté dans ce chapitre

La CPT présentée dans ce chapitre est basée sur le protocole à envoi de messages, car c'est celui utilisé dans nos deux implantations. Le choix du protocole fut évident pour ce qui est de l'implantation de CPT pour le multi-ordinateur, puisque celui-ci ne dispose pas de mémoire partagée. Dans le cas de l'ordinateur multi-processeur, le choix laisse place à la discussion puisque les deux protocoles peuvent être utilisés. On retrouve donc dans le chapitre 5, la justification de l'utilisation du protocole à envoi de messages malgré la présence d'une mémoire partagée.

3.3 Implantation de la CPT

Nous allons maintenant expliquer l'implantation de la CPT dans ParSubC. Cela permet de voir certains autres aspects et avantages de la CPT, et d'évaluer les coûts des différentes opérations de la CPT dans ParSubC.

3.3.1 Gestion et exécution des tâches

La présence de l'opérateur `!!` dans ParSubC a introduit le concept de tâche légère et lourde. La gestion de celle-ci se fait à travers les quatre étapes suivantes:

- Empilement d'une tâche légère
- Vol de tâche (et création de tâche lourde)
- Exécution d'une tâche volée
- Retour du vol de tâche

Les figures 3.3 à 3.8 présentent les structures de données utilisées et leurs états pour chacune des différentes étapes de la gestion de tâche. Ces figures ainsi que les quatre étapes sont présentées plus en profondeur dans les sous-sections qui suivent.

Tâche légère

Avec la CPT, le coût associé à la rencontre de l'opérateur `!!` est très petit car il n'y a pas immédiatement création de tâche, mais simplement empilement d'une tâche légère. Une tâche légère est constituée de deux éléments: un état et un pointeur `fp`. Le champ **`état`** décrit l'état de la tâche et peut prendre une des trois valeurs suivantes:

- *Inactive* – La tâche n'a été ni volée, ni exécutée, et est donc disponible pour exécution.
- *Volée* – La tâche a été volée, mais son exécution n'est pas terminée.
- *Terminée* – L'exécution de la tâche est terminée et le résultat, s'il y en a un, a été retourné à la victime.

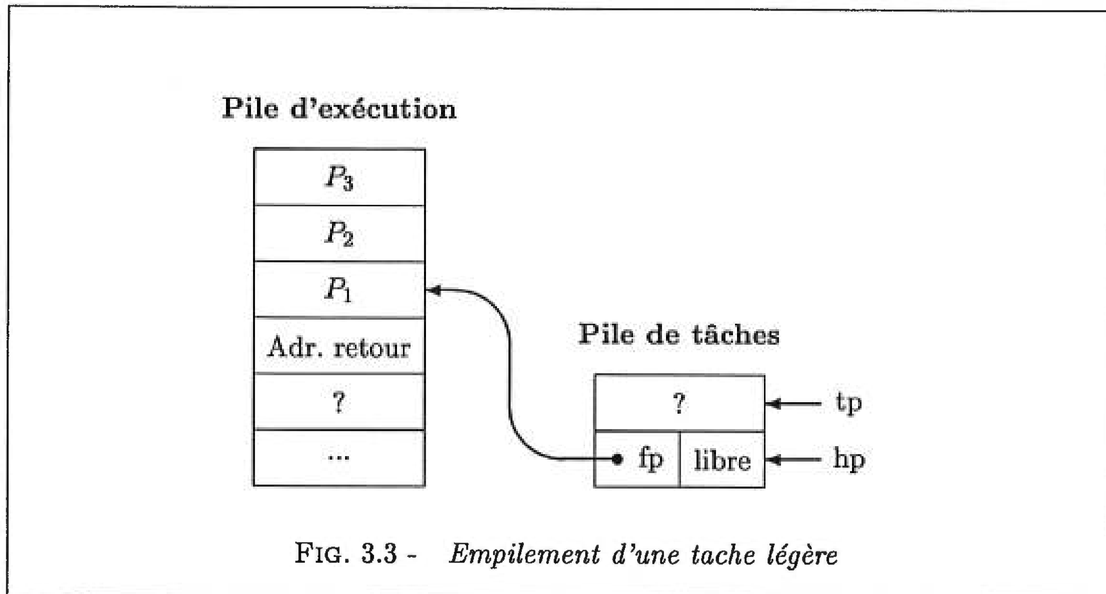

FIG. 3.3 - *Empilement d'une tache légère*

Comme on peut le voir dans la figure 3.3, lorsqu'une tâche légère est empilée sur la pile de tâches, le bloc d'activation de la fonction correspondante à la tâche est installé sur la pile d'exécution. Le premier élément que l'on retrouve dans la pile d'exécution est le résultat de la fonction. Comme la fonction n'a pas encore été exécutée, le résultat n'est pas encore déterminé et ceci est représenté par un point d'interrogation. Il est à noter qu'il n'y a pas d'espace réservé dans la pile pour le résultat si la fonction est de type `void`. L'élément suivant dans la pile d'exécution est l'adresse de retour `AR` de la fonction, suivie ensuite des éléments P_1, P_2 et P_3 qui sont les paramètres de la fonction (dans notre exemple, la fonction possède trois paramètres).

Le deuxième champ d'une tâche légère, soit `fp`, pointe sur le bloc d'activation de la fonction et permet de retrouver les informations nécessaires à la création de la tâche exécutable lors du vol de tâche. Les deux autres pointeurs que l'on retrouve dans la figure 3.3, soit `tp` et `hp`, servent à la gestion de la pile de tâches. Le pointeur `tp` ("tail pointer") permet d'empiler et de dépiler les tâches alors que `hp` ("head pointer") sert au vol de tâche. C'est le pointeur `hp` qui permet d'accéder à la structure comme une queue.

Comme le bloc d'activation de la fonction correspondante à la tâche légère est installé sur la pile d'exécution, s'il n'y a pas de vol de tâche alors, après l'exécution du bloc d'énoncés, il y aura un branchement au point d'entrée de la fonction et il n'y aura pas eu de création de tâche lourde. Le seul travail effectué est d'avoir empilé et dépilé un pointeur et assigner un état. Il n'y pas de suspension de tâche courante, puisque

```
__pfib: case 2:
   if ((*(int*)fp) < (2)) /* Est-ce que (i<2) ? */
   {
     *(int*)(fp-2) = *(int*)fp;
     goto _return; /* On retourne i */
   }
   *(int*)(fp+6) = (*(int*)fp)-1;
   fp += 6;
   *(int*)(fp-1) = 3;
   /* On empile une tache correspondant a pfib(i-1) */
   tp->state = INACTIVE;
   tp++->fp = fp;
   /* Appel de fonction pfib(i-2) */
   *(int*)(fp+4) = (*(int*)(fp-6))-2;
   fp += 4;
   *(int*)(fp-1) = 4;
   goto __pfib;
case 4:
   fp -= 4;
   (*((int*)(fp-4))) = (*(int*)(fp+2));
   /* Tache volee ? */
    if (tp <= hp) goto _stolen;
    --tp;
   goto __pfib; /* La tache n'a pas ete volee, alors
                   on l'execute */
case 3:
   fp -= 6;
   (*((int*)(fp+1))) = (*(int*)(fp+4));
   *(int*)(fp-2) = (*(int*)(fp+1))+(*(int*)(fp+2));
   goto _return;    /* On retourne t1+t2 */
```

FIG. 3.4 - *Code généré pour la fonction* `pfib`.

c'est tout simplement la tâche courante qui continue. Il y a donc très peu d'opérations associées à la gestion de tâche lorsqu'il n'y a pas de vol de tâche, et l'exécution de la fonction s'effectue alors comme une exécution sans ! !.

La figure 3.4 montre le code généré par le compilateur ParSubC pour la fonction `pfib` de la figure 2.4, qui est une variante parallèle de la fonction `fib` du chapitre 1. Dans cet exemple, on retrouve encadré le code C associé à la gestion des tâche dans le programme. On peut ainsi voir qu'empiler une tâche correspond à trois opérations

simples:

```
tp->state = INACTIVE;
tp->fp = fp;
tp++;
```

Et à la fin de l'exécution d'un bloc d'énoncés associée à l'opérateur `!!`, il faut vérifier si la tâche a été volée. Comme une nouvelle tâche est toujours mise sur le dessus de la pile et que les tâches sont volées par le dessous, on peut effectuer cette vérification par simple comparaison des pointeurs `hp` et `tp`. Dans la figure 3.4, on peut voir que le code C généré pour cette vérification est:

```
if (tp <= hp) goto _stolen;
--tp;
```

Il aurait également été possible pour vérifier si la tâche a été volée de faire le test `((tp-1)-->state == INACTIVE)`. La première méthode est cependant plus rapide à exécuter.

Vol de tâche

Lorsqu'il y a vol de tâche, il faut, à partir des informations contenues dans la tâche légère, être capable de créer une tâche exécutable (ou lourde). La seule information disponible dans la tâche légère qui puisse nous permettre cela est le pointeur `fp`. Il faut donc à partir de ce pointeur être capable de retrouver les informations nécessaires à la création de la tâche lourde qui sont:

- Identificateur de tâche (TID pour "Task ID").
- Adresse de retour de la fonction associée à la tâche.
- Paramètres de la fonction associée à la tâche.

Comme le champ `fp` de la tâche légère pointe sur la pile d'exécution, il nous permet de retrouver les arguments de la fonction ainsi que l'adresse de retour. Pour ce qui est du TID, il correspond à l'adresse mémoire où se situe la tâche légère, qui est unique pour chaque tâche (autant dans l'implantation avec mémoire partagée que dans celle avec mémoire distribuée). Le TID permet aussi de reconnaître la fin de l'exécution d'un programme, car on donne à la tâche correspondante à la fonction `main` un TID de valeur 0.

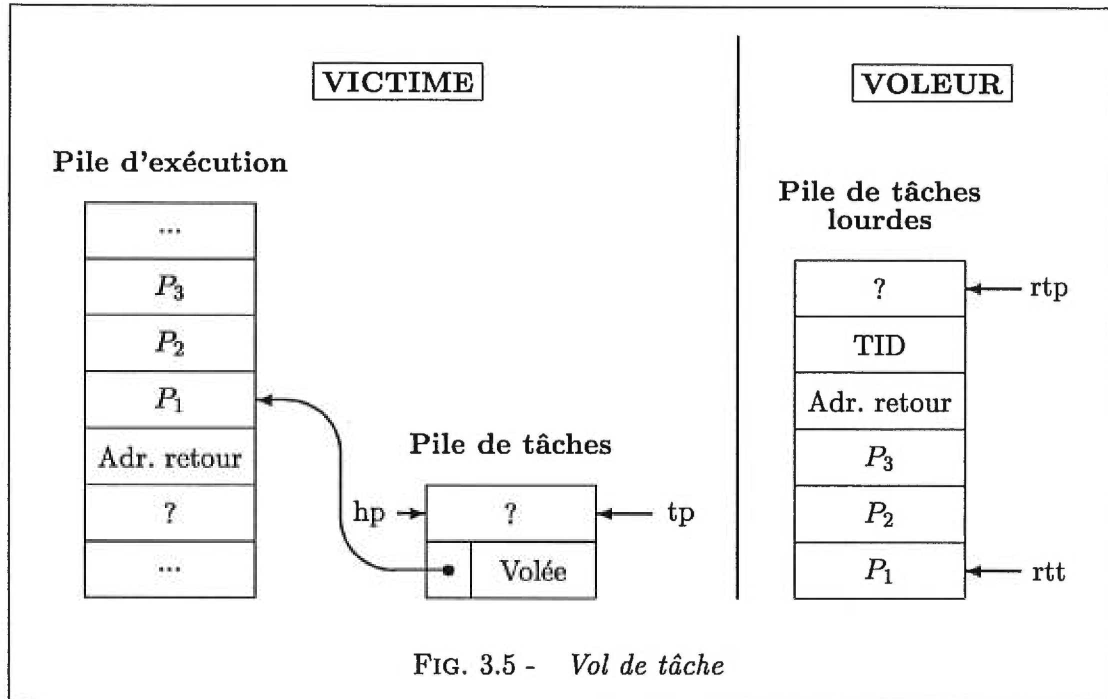

FIG. 3.5 - *Vol de tâche*

Cependant, bien que la tâche légère permette, grâce au pointeur `fp`, de retrouver où sont situés les paramètres de la fonction, elle n'en indique pas le nombre. Cette information nécessaire à la création de la tâche lourde, est gardée dans un tableau appelé `_control_points` créé à la compilation et dont chaque processeur possède une copie identique. Un exemple de ce tableau est montré dans la figure 3.6. Pour trouver le nombre de paramètres de la fonction, il suffit d'indexer le tableau des points de contrôle avec l'adresse de retour pointée par le pointeur `fp` de la tâche légère. Cette indexation nous retourne un triplet contenant le nombre de paramètres, le point d'entrée ainsi que la taille du résultat de la fonction. Comme nous le verrons plus loin, ces deux derniers champs sont utiles pour l'exécution de la fonction.

Une particularité des adresses de retour qui nous permet de les utiliser comme index, est qu'elles sont toujours uniques. Il n'y aura jamais deux appels de fonction qui se termineront au même endroit dans un programme. Il est important ici de rappeler que, telles qu'elles ont été présentées dans la section 1.5, les adresses de retour correspondent à des `case` de la commande `switch` et donc, elles s'utilisent beaucoup mieux comme index que de vraies adresses. L'utilisation du tableau `_control_points` est une technique simple et très rapide pour retrouver l'information complémentaire nécessaire à la création et l'exécution des tâches lourdes.

```
int _control_points[][3] =
{  /* {Point d'entree, Nombre d'arguments, Taille du resultat. } */
 /*0*/{5,0,0}
,/*1*/{0,0,0}
,/*2*/{0,0,0}
,/*3*/{2,1,1}
,/*4*/{2,1,1}
,/*5*/{0,0,0}
,/*6*/{1,2,0}
,/*7*/{2,1,1}
};
```

FIG. 3.6 - *Points de contrôle de* `pfib.c`

Après avoir été créée, la tâche lourde est ensuite copiée dans la *pile de tâches lourdes* du voleur abrégée par RTS pour "Ready Task Stack" (voir la figure 3.5). La victime reflète ce vol de tâche en changeant l'état de la tâche légère dans sa pile de tâches pour *Volée.*

On peut se demander pourquoi utiliser une pile pour recevoir la tâche lourde puisqu'avec le vol de tâche tel qu'il a été présenté dans ce chapitre, un processeur ne peut avoir plus d'une tâche dans sa pile de tâches lourdes. Le réponse se trouve dans le chapitre 4 qui porte sur l'implantation pour mémoire distribuée et où on y présente la migration de tâche. Avec la migration de tâche, nous verrons qu'il est possible pour un voleur d'avoir plusieurs tâches dans sa pile de tâches lourdes.

Exécution d'une tâche volée

Comme dans ParSubC une tâche correspond en fait à un appel de fonction, l'exécution d'une tâche sera semblable à un appel de fonction. On retrouve dans la figure 3.7 toutes les informations nécessaires à l'exécution d'une tâche qu'il faut avoir dans la pile d'exécution. On y retrouve tous les éléments d'un appel de fonction standard: une adresse de retour, les arguments de la fonction et, si nécessaire, un espace pour le résultat de la fonction. La seule différence entre l'exécution d'une tâche et un appel de fonction est le TID nécessaire dans le cas d'une tâche. Ce TID servira à mettre à jour la tâche légère lorsque l'exécution de la tâche lourde correspondante sera terminée.

Lorsqu'un processeur réussit un vol de tâche, il doit donc copier le champ TID de la tâche lourde contenue dans son RTS dans sa pile d'exécution. Il doit ensuite laisser l'espace pour le résultat de la fonction, s'il y en a un, mais pour cela il doit en connaître la taille. Comme cette information est contenue dans le tableau de points de contrôle, le

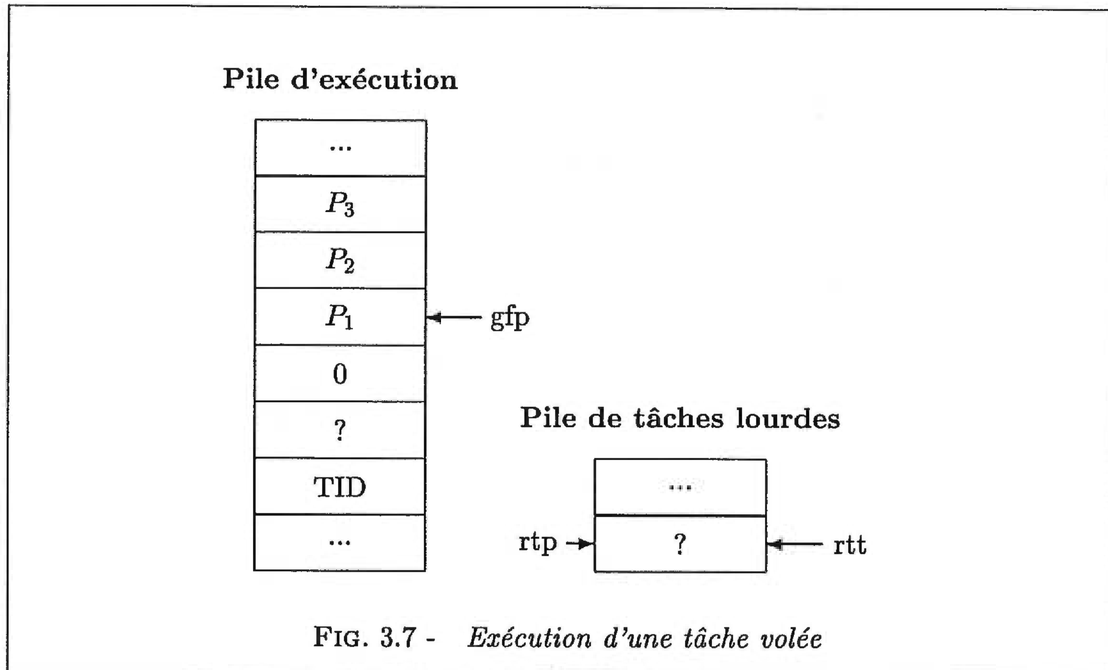

FIG. 3.7 - *Exécution d'une tâche volée*

voleur n'a qu'à indexer ce tableau avec l'adresse de retour contenue dans la tâche lourde pour la retrouver.

Il est à noter que l'adresse de retour contenue dans la pile d'exécution n'est pas celle contenue dans la tâche lourde mais plutôt 0. La raison est que lorsque la fonction associée à la tâche lourde se termine, on ne veut pas que le voleur continue l'exécution à l'adresse de retour, mais on veut plutôt qu'il y ait retour de vol de tâche. C'est ce qui arrivera avec l'adresse de retour 0, puisque l'adresse 0 (où plutôt le `case 0`) contient le code de retour de vol de tâche.

Les dernières informations à copier dans la pile d'exécution sont les variables locales de la fonction, qui sont contenues dans la tâche lourde. Ensuite, pour débuter l'exécution, le voleur n'a qu'à brancher au point d'entrée contenu dans le triplet du tableau de points de contrôle correspondant à l'adresse de retour de la fonction. Comme pour les adresses de retour, dans ParSubC, les points d'entrée correspondent à des `case` de l'énoncé `switch` principal.

Retour de vol de tâche

Lorsque le voleur a terminé l'exécution d'une tâche volée, le résultat de la fonction, s'il y en a un, est alors copié dans la pile d'exécution de la victime dans l'espace réservé à

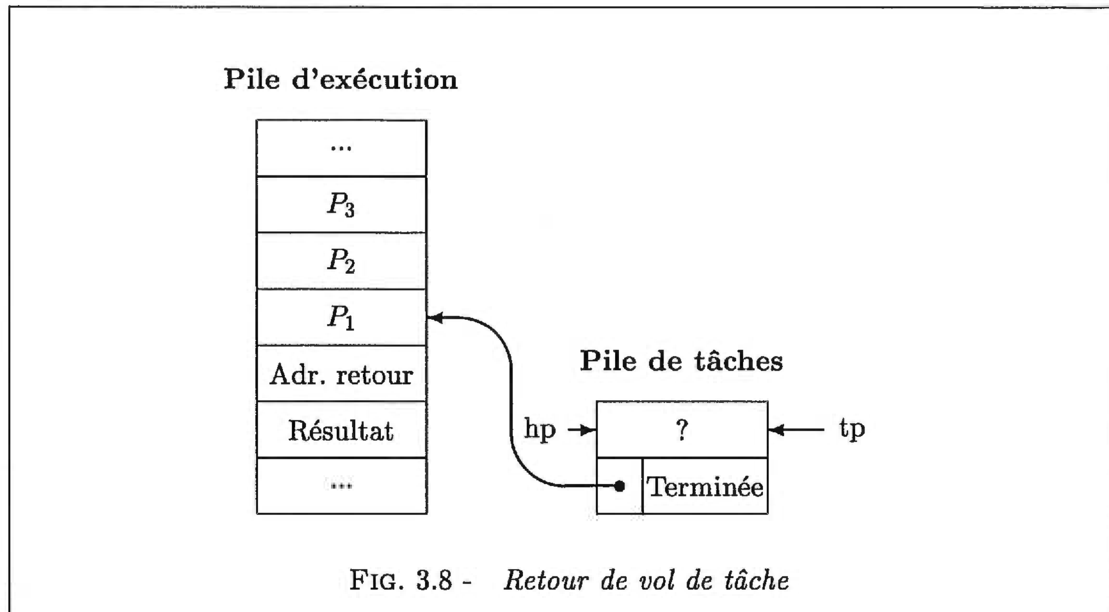

FIG. 3.8 - *Retour de vol de tâche*

cette fin. La tâche légère contenue dans la pile de tâches de la victime est également mise à jour pour refléter son nouvel état, soit *Terminée*. La figure 3.8 contient un exemple de tâche terminée.

Il est à noter qu'avant d'effectuer le retour de vol de tâche, le voleur doit auparavant vérifier s'il le TID de la tâche n'est pas de 0, car cela signifie alors la fin de l'exécution de la fonction `main` (et donc du programme).

3.3.2 Recherche de travail

Dans ParSubC, ce sont les processeurs libres qui ont la responsabilité de se trouver du travail. Nous allons maintenant définir précisément quand un processeur est libre.

Processeur libre

Si un processeur n'a aucune tâche dans sa pile d'exécution (comme c'est le cas au début de l'exécution d'un programme pour tous les processeurs sauf un), il ne peut non plus avoir de tâches dans sa pile de tâches légères puisqu'une tâche légère pointe toujours sur le bloc d'activation de la fonction correspondante dans la pile d'exécution. C'est donc un cas trivial où un processeur est libre.

Lorsqu'un processeur termine l'exécution d'un bloc d'énoncés et que la tâche associée à ce bloc a été volée, le processeur doit suspendre sa tâche courante et devient alors libre. Cette tâche devra rester suspendue jusqu'à ce que l'exécution de la tâche volée soit terminée. Ainsi, un processeur est libre lorsqu'il remplit une des deux conditions suivantes:

- Il n'a pas de tâche dans sa pile d'exécution (et donc pas de tâche dans la pile de tâche).
- L'exécution de la tâche courante doit être suspendue (à cause du vol de la tâche du dessus de la pile).

Recherche de travail

Il y a deux sources de travail à regarder pour un processeur libre. Premièrement, si le processeur libre a dû suspendre une tâche, il doit régulièrement vérifier si l'exécution de la dernière tâche volée est terminée, car c'est celle-ci qui oblige à la suspension de la tâche courante. Lorsque son exécution sera terminée, le processeur pourra alors reprendre l'exécution de la tâche suspendue. Reprendre l'exécution de cette tâche se fait tout simplement en sautant à l'adresse pointée par le champ `fp` de la tâche volée.

L'autre source de travail provient bien sûr des autres processeurs. Comme le protocole d'accès aux piles utilisé dans notre implantation est celui à envoi de messages, pour arriver à voler une tâche, le processeur doit envoyer des requêtes de vol. Il en envoie à un seul processeur à la fois, et attend la réponse de celui-ci avant d'en envoyer une autre. Si la réponse est négative, il passe à un autre processeur et continue cycliquement jusqu'à ce qu'il ait trouvé du travail. Le code C associé à la recherche de travail est présenté dans la figure 3.9.

Au début de l'exécution du programme, les processeurs commencent tous leur recherche de travail avec le processeur 0. La raison est qu'au début, seul le processeur 0 peut avoir des tâches disponibles. Il n'aura peut-être pas le temps de créer des tâches pour tous les processeurs, dépendant du programme, mais il est le seul qui peut en avoir. Si le nombre de processeurs utilisés est très grand (et donc le processeur 0 a beaucoup de requêtes à traiter), il se peut qu'un autre processeur ait eu le temps de voler une tâche et de commencer à l'exécuter au point d'avoir une tâche disponible. Mais avec un nombre relativement petit, 16 processeurs sont utilisés pour ce travail, cela est peu probable et le processeur 0 est donc celui qui a le plus de chance d'avoir des tâches disponibles au début de l'exécution.

```
int get_work()
{
  loop:

  /* Redemarrer la tache courante, si la tache du dessus de la
     pile est prete. */
  if ((tp-1)->state == TERMINATED)
  {
    if (hp == tp) hp--;
    tp--;

    /* On retourne un pointeur a l'endroit ou l'execution doit
       reprendre. */
    return *(int *)(fp-1);
  }

  /* On verifie si une tache est disponible dans la pile de taches
     pretes. */
  if (rtp != rtt)
  {
    int i;
    word **tid;
    int ra, pc, n1, n2;

    tid = *(word ***)(--rtp);       /* Identificateur de tache */
    ra = *(int *)(--rtp);           /* Adresse de retour */
    pc = _control_points[ra][0];    /* Point d'entree */
    n1 = _control_points[ra][1];    /* Nombre d'arguments */
    n2 = _control_points[ra][2];    /* Taille du resultat */

    /* On installe la tache dans la pile d'execution */
    *(word ***)(fp++) = tid;
    fp += n2;
    *(int *)(fp++) = 0;
    rtp -= n1;
    for (i=0; i<n1; i++) fp[i] = rtp[i]; /* Copie des arguments */

    return pc; /* On retourne le point d'entree */
  }

  /* S'il n'y a pas deja une requete de vol en attente de
     traitement, on en envoie une. */
  if (pending_steal_id == -1)
  {
    do
      /* On passe a la prochaine victime. */
      victim_id = (victim_id + 1) %_nb_nodes();
    while (victim_id == _self());
    pending_steal_id = victim_id;
    SEND( pending_steal_id, STEAL_MSG );
  }

  goto loop;
}
```

FIG. 3.9 - *Fonction de recherche de travail* (get_work)

Lorsqu'un processeur a terminé l'exécution d'une tâche ou a du suspendre son exécution, il recommence sa recherche de travail au processeur suivant le dernier qui lui a donné du travail. On aurait pu également repartir au processeur 0, mais cela aurait pour effet de vider la pile de tâches du processeur 0 beaucoup plus rapidement que les autres, et celui-ci se retrouverait à son tour sans tâche. En plus, avec un nombre élevé de processeurs cela pourrait avoir pour effet d'engorger le processeur 0 de requêtes de vol.

Une autre approche qu'un processeur libre aurait pu utiliser comme méthode de recherche de travail aurait été de commencer par le dernier processeur qui lui a fournit du travail. L'avantage de cette approche est que les chances que ce dernier ait une tâche sont potentiellement plus élevées puisque précédemment il en avait. Cependant, on risque ici aussi de vider prématuremment la pile de tâches de ce dernier.

Partir du processeur suivant le dernier qui a fournit du travail a donc l'avantage de balancer les vols de tâches ce qui permet:

- D'éviter d'engorger un processeur avec un nombre élevé de requêtes de vol.
- De puiser dans les piles de tâches de façon plus équilibrée.

3.4 Résumé

La *création paresseuse de tâche* est un mécanisme de partitionnement dynamique où ce sont les processeurs libres qui ont la responsabilité de se trouver du travail à l'aide du principe de vol de tâche. Ce n'est seulement qu'au moment de ces vols de tâches que les tâches exécutables (ou lourdes) sont créées, et donc si une tâche n'est pas volée, elle ne sera jamais créée puisque ce n'est qu'un appel de fonction normal qui a alors lieu. Cela permet ainsi de réduire le temps associé à la gestion des tâches et fait de la CPT un mécanisme de partitionnement peu coûteux.

Notre implantation de la CPT utilise trois structures de données pour les tâches, soit: une pile d'exécution, une pile (ou queue) de tâches légères et une pile de tâches lourdes. Les tâches légères n'étant constituées que du minimum d'information nécessaire à la création des tâches exécutables, le surcoût associé à la gestion des tâches non-volées est très petit. Celui associé à la gestion des tâches volées a moins d'influence car il est compensé par le parallélisme offert, et son impact dépend du moyen de communication utilisé par les processeurs pour s'échanger les tâches.

Ce qui permet de rendre minime la quantité d'information à empiler sur la pile de tâches est l'utilisation du tableau de points de contrôle qui contient certaines informations complémentaires nécessaires à la création des tâches. Ces données, étant dans un tableau créé à la compilation, n'ont pas besoin d'être empilées avec les tâche. Lors d'un vol de tâche, ce tableau est donc consulté afin de pouvoir créer la tâche qui est alors mise dans une pile de tâches prêtes. De là, le voleur pourra, encore à l'aide du tableau de points de contrôle, débuter l'exécution de la tâche.

Chapitre 4

Implantation avec mémoire distribuée

Ce chapitre présente et discute de l'implantation d'un compilateur ParSubC pour un multi-ordinateur à mémoire partagée, constitué d'un réseau de stations de travail. La principale particularité et difficulté rencontrées avec cette implantation est l'utilisation du réseau pour les communications inter-processeurs pour pallier à l'absence d'une mémoire partagée. Le langage ParSubC ayant été défini pour offrir un modèle de programmation à mémoire partagée, une mémoire partagée virtuelle a été implantée à l'aide du réseau pour répondre à ce besoin. De plus, les communications inter-processeurs sont également nécessaires pour l'implantation de la CPT, puisque les stations de travail doivent être capables de s'échanger des tâches entre elles pour permettre le partitionnement.

4.1 Notation utilisée dans ce chapitre

Dans le chapitre 3, plusieurs structures de données et pointeurs ont été introduits (pile d'exécution, pile de tâches et pile de tâches prêtes, etc.). Comme ces structures et pointeurs seront référencés dans plusieurs exemples dans ce chapitre, il serait intéressant de définir les différents identificateurs associés à chacun d'eux. La figure 4.1 montre toutes les structures de données utilisées ainsi que les pointeurs s'y rattachant. On peut remarquer deux éléments importants sur la notation utilisée pour ces identificateurs: ils sont tous préfixés du caractère _ et certains pointeurs sont également préfixés de la lettre *g*. Le préfixe _ permet de distinguer les pointeurs de piles reliés à la CPT de pointeurs

Pile d'exécution	
`_stk`	Pile d'exécution
`_fp`	Pointeur sur la tâche en exécution
`_gfp`	Pointeur global de `_fp`
Pile (ou queue) de tâches	
`_ltq`	Pile de tâches
`_hp`	Pointeur sur la prochaine tâche à voler
`_tp`	Pointeur sur l'espace de la prochaine tâche à empiler
`_gtp`	Pointeur global de `_tp`
Pile de tâches prêtes	
`_rtq`	Pile de tâches prêtes
`_rtp`	Pointeur sur la fin de la pile
`_rtt`	Pointeur sur l'espace de la prochaine tâche volée

TAB. 4.1 - *Structures de données et pointeurs utilisés dans le code*

de l'usager.

Le préfixe *g*, quant à lui, sert à distinguer les pointeurs locaux utilisés dans la fonction `_execute` et ceux globaux utilisés à l'extérieur de la fonction. Telle que mentionnée au chapitre 1, la fonction `_execute` est la fonction qui contient le code du programme ParSubC après la compilation de celui-ci. Comme les pointeurs de la CPT sont référencés dans plusieurs fonctions, ils sont gardés dans des variables globales. Mais par souci d'efficacité, il existe des variables locales à la fonction `_execute`, correspondant aux variables globales `_gfp` et `_gtp` et annotées `register` à leur déclaration. Cette annotation indique au compilateur C que ces variables peuvent être allouées dans des registres, ce qui permet d'accéder à ces variables très rapidement. Comme ces variables sont souvent utilisées, surtout `_fp`, cela résulte en de meilleures performances. Cependant, pour garder la cohérence entre les variables locales `_fp` et `_tp` et les variables globales `_gfp` et `_gtp`, les pointeurs globaux sont mis-à-jour avant chaque appel de fonction fait par la fonction `_execute` et les variables locales le sont à chaque retour d'appel de fonction fait par `_execute`.

4.2 Architecture de l'implantation

Pour faciliter la présentation de l'implantation de ParSubC pour un multi-ordinateur à mémoire distribuée, celle-ci sera brisée en plusieurs composantes qui représenteront

différentes fonctionnalités requises pour l'exécution parallèle de programmes ParSubC. La figure 4.1 présente les différentes composantes de notre implantation en plus des interactions entre eux. Il est à noter, que quoique ces composantes soient très distinctes dans la figure, elles ne le sont pas si clairement dans le code réel. Par exemple, les composantes *programme ParSubC* et *CPT* ne sont pas complètement distinctes; ainsi une fonction correspondant à la composante *programme ParSubC* peut contenir certains éléments de la composante *CPT*.

Une première composante, que l'on retrouve dans le haut de la figure 4.1, correspond à un programme écrit dans le langage ParSubC. Ce programme est compilé en un programme C de manière à ajouter au programme initial certains éléments lui permettant d'interagir avec les composantes CPT et *mémoire partagée virtuelle* (MPV). Ces éléments lui permettent de partitionner son travail en utilisant les mécanismes de la CPT en plus de pouvoir accéder à la mémoire partagée virtuelle. Les interactions avec ces deux autres composantes sont donc essentielles à l'exécution parallèle d'un programme ParSubC. Le programme C est en quelque sorte la composante maître du système puisqu'elle constitue les traitements spécifiés par l'usager et en plus c'est celle qui dirige les opérations. Les autres composantes peuvent être vues comme étant des outils supportant l'exécution parallèle du programme ParSubC.

Pour permettre le bon fonctionnement des composantes CPT et MPV, il leur faut pouvoir accéder au réseau, car elles doivent pouvoir transférer de l'information d'une station de travail à une autre. Les deux autres composantes présentes dans le système, soit les composantes *Interface au réseau* et *Distributeur de messages*, ont comme fonction de gérer les messages qui circulent entre les stations. La première composante, comme son nom l'indique, agit comme interface au réseau. Elle permet entre autre d'envoyer des messages sur le réseau. Ainsi, lorsque les composantes CPT et MPV veulent envoyer un message à une autre station, elles n'ont qu'à communiquer leur message ainsi que le destinataire du message à la composante IR. De plus, c'est elle qui s'occupe de recevoir les messages du réseau.

Comme la composante *Interface au réseau* n'est qu'un interface au réseau, lorsqu'elle reçoit un message, elle ne fait aucun traitement du contenu du message. Le contenu du message est passé à la composante *Distributeur de messages* qui a la responsabilité d'acheminer le message à la composante appropriée (CPT ou MVP) afin que le message soit correctement traité.

Dans les sections qui suivent, la fonctionnalité de chaque composante sera revue plus en profondeur, en plus d'y voir l'implantation.

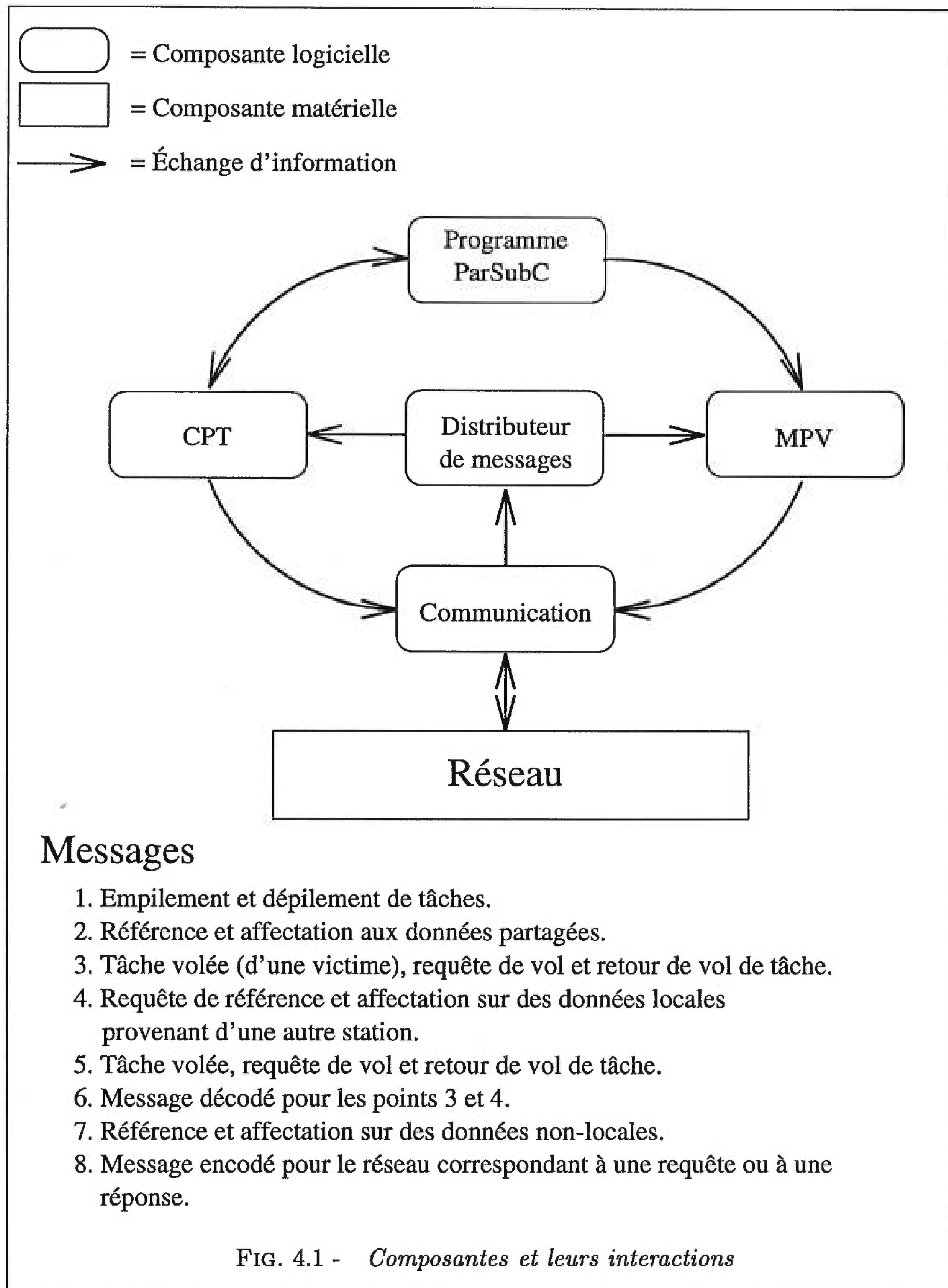

FIG. 4.1 - *Composantes et leurs interactions*

4.3 Programme ParSubC

La première composante que nous regarderons est le programme ParSubC, car c'est en quelque sorte l'élément directeur du programme total. Pour que cette composante puisse interagir avec les autres, il faut premièrement la compiler et générer du code qui contiendra les interactions avec les composantes. Le compilateur utilisé est écrit en Scheme mais comme mentionné et discuté dans le chapitre 1, le code généré est du C. C'est ce code C qui contient le programme ParSubC décomposé en expressions de base auxquelles on a ajouté les différents éléments permettant le partitionnement et les accès à la mémoire partagée virtuelle.

La compilation du programme ParSubC vers du C permet donc d'établir les liens entre les différentes composantes du système. Ce nouveau programme C est ensuite compilé, ainsi que toutes les autres composantes, pour être ensuite liés ensemble pour créer un fichier exécutable. Le programme ParSubC est alors prêt pour l'exécution parallèle.

4.4 Communications

Avec une architecture à mémoire partagée, la communication entre les processeurs est très facile à réaliser, car la mémoire partagée constitue un support de communication rapide et simple à utiliser. Malheureusement, notre multi-ordinateur ne possède pas une telle mémoire, ce qui oblige donc, pour implanter ParSubC pour ce type d'ordinateur, à concevoir une composante permettant la communication inter-processeurs. Cette composante est essentielle pour le partitionnement de la CPT ainsi que pour l'implantation de la mémoire partagée virtuelle.

L'implantation des communications se divise en deux parties, soit une composante qui permet d'envoyer et de recevoir des messages du réseau de stations de travail, et une autre, de plus haut niveau, qui permet de traiter les différents messages reçus selon leur contenu.

4.4.1 Interface au réseau

Lors du lancement de l'exécution d'un programme ParSubC, une copie du programme est premièrement démarrée sur la station qui a lancé l'exécution. Ensuite, cette première station, identifiée comme étant la station 0, a la responsabilité de démarrer ce

programme sur les autres stations. Le réseau est évidemment utilisé pour réaliser cela, mais tout est fait de façon transparente puisqu'il existe déjà un utilitaire Unix, `rsh`, qui permet, à partir d'une station, de faire exécuter un programme sur une autre station. Nous avons maintenant des stations qui exécutent un programme commun et qui sont reliées en réseau. À partir de ce point, il faut maintenant définir de quelle façon les stations utiliseront le réseau pour communiquer.

Le protocole de communication utilisé pour les stations de travail est TCP/IP ([Dig93], [Hew92]). C'est un des protocoles les plus répandus, et des librairies sont donc disponibles sur la plupart des stations Unix. C'est un avantage important car cela permet de garder notre implantation facilement portable. L'implantation actuelle est compatible avec un réseau de stations DEC, Sun et HP.

En plus de la portabilité, le protocole TCP/IP possède des caractéristiques qui sont indispensables à l'implantation de ParSubC. C'est un protocole qui est dit fiable puisqu'il détecte et corrige les erreurs de transmission ainsi que les messages perdus. Losqu'un message est envoyé, TCP/IP garantit donc que le message arrivera à destination et ce sans erreur. C'est évidemment là une caractéristique essentielle pour implanter ParSubC, sinon c'est la fiabilité même des programmes ParSubC qui serait atteinte. Une autre caractéristique essentielle retrouvée avec le protocole TCP/IP est que l'ordonnancement des messages à l'envoi est préservé à l'arrivée. Ainsi, lorsqu'un lien de communication TCP/IP est établie entre deux stations de travail, il est garantit qu'un message envoyé avant un autre sur ce lien arrivera également avant.

Cependant, il est important de noter que pour préserver l'ordonnancement des messages, les liens TCP/IP établis entre les stations doivent être préservés pendant toute la durée du programme. Si un lien TCP/IP est fermé et rétabli, le nouveau lien pourrait avoir un chemin plus court, et il y aurait donc possibilité qu'un message envoyé sur le nouveau lien arrive avant un message envoyé sur l'ancien lien (qui se fait à travers un chemin plus long). L'ordonnancement des messages est une caractéristique indispensable pour, par exemple, des affectations à la mémoire partagée car changer l'ordre des affectations peut empêcher la bonne exécution de beaucoup de programmes.

Communication basée sur les interruptions

Lorsqu'une station reçoit un message, il est préférable qu'elle s'en aperçoive le plus rapidement possible, car tel qu'expliqué plus loin, certains des messages reçus sont des requêtes envoyées par des processeurs qui sont en attente d'une réponse. Plus le temps de réponse est petit, plus vite les processeurs pourront reprendre leur travail. Par souci

d'efficacité, il est préférable que le temps entre l'envoi d'une requête et la réception de la réponse soit aussi petit que possible.

Pour arriver à détecter rapidement la réception de messages, l'implantation des communications est basée sur les interruptions matérielles. Une routine d'interruption est associée au signal d'interruption SIGIO (qui est le signal correspondant aux entrées/sorties), et c'est cette routine qui permet de traiter le message reçu immédiatement. Le traitement effectué dépend du type de message reçu, et pour cela la routine d'interruption donne le message reçu au distributeur de message, mais ceci est expliqué plus en profondeur dans la section suivante.

Il aurait également été possible d'utiliser le "polling" comme technique d'interruption. L'avantage que possède cette technique sur les interruptions matérielles est d'être uniquement logicielle, ce qui lui permet d'avoir un temps de déclenchement plus rapide. Cependant, comme le "polling" consiste en des vérifications périodiques, il peut y avoir un délai entre la réception d'un message et le déclenchement de l'interruption, contrairement aux interruptions matérielles qui sont déclenchées immédiatement. De plus, avec ces dernières, il n'y a aucun coût associé à faire des vérifications inutiles comme il peut arriver avec le "polling". Les interruptions matérielles ont donc étés choisies pour leur rapidité à être déclenchées et le fait qu'il n'y a aucun coût lorsqu'il n'y a pas de messages reçus.

Sections critiques

Malgré qu'il soit préférable pour l'efficacité des programmes que les messages reçus soit traités immédiatement, il existe dans l'implantation de ParSubC certaines sections critiques pendant lesquelles il ne doit pas y avoir de traitement des messages car cela pourrait nuire à la bonne exécution de certaines opérations. Il faut donc être en mesure de retarder pour une période donnée le traitement des messages reçus. Pour cela, avant de traiter un message, la routine d'interruption consulte une variable qui lui indique si le traitement du message reçu doit se faire immédiatement ou s'il doit être différé. À la fin de la section critique, il y a vérification d'une autre variable pour déterminer si des messages ont été reçus, et dans l'affirmative ces messages sont immédiatement traités. Les sections critiques présentées dans ce document sont toujours insérées entre une paire d'énoncés comme suit:

```
ASYNC_RECV_DISABLE();
<section critique>
```

```
typedef struct msg_h
{
  char   msg_id; /* Identification du message */
  char   *adr;   /* Pointeur */
  double val;    /* Tampon */
  int    extra;  /* Espace supplementaire */
} msg_header;
```

FIG. 4.2 - *Éléments composants un message*

```
ASYNC_RECV_ENABLE();
```

Plus précisément, ces deux primitives sont des macros C qui sont définies comme suit:

```
#define ASYNC_RECV_DISABLE(){ com_async_recv_allowed = 0; }
#define ASYNC_RECV_ENABLE(){ com_async_recv_allowed = 1;
        if (com_async_recv_pending) com_async_recv_handler(); }
```

Le retardement du traitement des messages peut résulter en plusieurs messages non-traités, d'où la nécessité d'avoir une queue pour garder ces messages. Ce système de queue est déjà fourni par TCP/IP, et cela est en quelque sorte transparent dans l'implantation de ParSubC. Il a cependant fallu voir à ce que la taille de la queue, qui est un paramètre variable de TCP/IP, soit suffisante pour les besoins de ParSubC.

4.4.2 Distributeur de messages

Dans cette implantation de ParSubC, c'est le réseau qui permet aux stations de communiquer entre elles. Cette communication est essentielle pour permettre le partitionnement ainsi que pour implanter la mémoire partagée virtuelle. Lors de l'exécution d'un programme, les stations pourront s'échanger des messages leur permettant de partitionner le travail. Pour être plus précis, la CPT requiert des messages de requête de vol ainsi que de retour de vol. Pour la MPV, les messages serviront plutôt à faire des accès (lecture et écriture) à des variables non-locales à la station qui y accèdent. La tâche de notre distributeur de messages sera donc de reconnaître le type de message qu'il reçoit, pour ainsi prendre les actions appropriées.

Pour simplifier l'implantation des communications, un seul format de message a été défini. Par souci d'efficacité, le format choisi est de petite taille mais permet de contenir les informations de toutes les requêtes possibles. De cette façon, toutes les

requêtes pourront être envoyées en un seul message, ce qui est plus performant. Les seuls éléments qui ne peuvent pas être inclus dans le format sont ceux qui possèdent une taille variable, par exemple une structure ou les arguments d'une tâche volée. Ainsi lorsqu'une structure devra être envoyée d'une station à une autre, elle devra donc être contenue dans un message à part. La figure 4.2 présente les différents champs de la structure correspondant au format du message utilisé dans notre implantation. Le premier champ, nommé `msg_id`, permet au distributeur d'identifier le message reçu (est-ce une requête de vol, un accès à la MPV, ...). Les autres champs ont différents usages dépendant du type de message envoyé.

Un premier exemple de requête qu'une station peut envoyer à une autre, est une référence à une variable non-locale. Le message envoyé devra évidemment contenir un identificateur de message correspondant à cette opération, mais aussi l'adresse de la variable désirée (contenue dans le champ `adr`). La réponse à ce message est un message qui contient la valeur de la variable référencée dans le champ `val`. Le champ `msg_id` permet de s'assurer que le message reçu est bien la réponse à sa requête.

Malgré que le champ `val` soit de type `double`, il pourra contenir tous les différents types C de base. Le seul motif pour avoir donné ce type à `val` est qu'il a la plus grande taille et peut donc contenir tous les autres types de base. Le champ `val` doit donc plutôt être vu comme un tampon ayant la taille d'un `double`.

Un autre exemple de message est lorsqu'une victime envoie une tâche à un voleur. Telle que mentionnée dans le chapitre sur la CPT, une tâche peut être définie par un identificateur de tâche (`tid`), une adresse de retour ainsi que les paramètres d'entrée. Le message envoyé par la victime sera donc le suivant:

`msg_id`	= `RESPONSE_STEAL_TASK_MSG`
`adr`	= `tid`
`val`	= *Non utilisé*
`extra`	= Adresse de retour

Les paramètres d'entrée ne peuvent être envoyés dans le message car la taille de paramètres n'étant pas constante, un format fixe n'est pas adéquat. Ceux-ci seront donc transférés dans un message à part, qui suivra le message d'identification. Après avoir reçu ce message, un voleur s'attend donc à recevoir un autre message correspondant aux paramètres. Grâce à l'adresse et au tableau de points de contrôle, le voleur est en mesure de connaître la taille des paramètres, et est donc en mesure de savoir quelle sera la taille du second message qu'il recevra.

Toutes les autres requêtes que peuvent envoyer les stations le sont de façon similaire

```
void f1()                          void f2(int *ptr)
{                                  {
  int a;                             ...
                                     *ptr = 0;
  ...                                ...
  f2(&a) !! { ... };               }
  ...
}
```

FIG. 4.3 - *Accès partagé à une variable locale*

aux deux exemples que nous venons de voir. Il n'est pas vraiment pertinent de passer à travers chacune d'elles puisque le mécanisme utilisé est le même: une requête est toujours identifiée grâce au `msg_id`, qui permet au distributeur d'agir en conséquence, et si une requête comporte des données de taille variable, alors celles-ci sont envoyées dans un message à part immédiatement après le message contenant la requête. Les sections sur la CPT et la MPV apporteront plus d'information sur les messages qui leurs sont propres.

4.5 Mémoire partagée virtuelle

Le langage ParSubC offre un modèle de programmation à mémoire partagée, et ce pour la portabilité du langage ainsi que pour la simplicité provenant de ce modèle de programmation. Avant de parler de l'implantation de cette mémoire partagée virtuelle (MPV), il serait intéressant de discerner quels types de variables de ParSubC peuvent être *partagés*, c'est-à-dire qui peuvent être accédés par plusieurs stations. De cette façon, il sera plus facile d'évaluer quel but notre MPV doit atteindre.

Le premier type de variable qui est partagé est celui des variables globales, car une variable globale est accessible à n'importe quel endroit dans un programme. Pour supporter un modèle de programmation à mémoire partagée, il faut donc permettre à toutes les stations d'avoir accès aux variables globales.

À première vue, les variables locales ne semblent pas pouvoir être partagées puisque la portée de celles-ci est au plus limitée à la fonction dans laquelle elles sont définies. En rappelant qu'une tâche correspond toujours à un appel de fonction, on peut affirmer que la portée d'une variable locale ne peut dépasser celle d'une tâche. Le problème cependant est qu'il est possible d'accéder à l'emplacement mémoire de celle-ci par des pointeurs. Dans l'exemple de la figure 4.3, si la tâche correspondant à la fonction `f2` se fait voler, alors le pointeur ne pointera pas sur une donnée locale puisque la variable a

se situe dans la mémoire de la victime. Avec les pointeurs, l'emplacement mémoire de toute variable, locale ou non, peut être accédée, et donc partagée par toutes les stations.

Pour supporter un modèle de programmation à mémoire partagée, notre MPV devra donc voir à ce que les stations puissent toutes accéder aux variables globales et devra également tenir compte du fait qu'un pointeur puisse contenir l'adresse d'une variable qui peut être contenue dans la mémoire de n'importe quelle station.

4.5.1 Variables partagées: copie unique *vs* copies multiples

Pour que plusieurs stations puissent partager une même variable, deux approches sont envisageables. La première approche est de n'avoir qu'une seule copie de la variable dans la mémoire d'une station, que l'on qualifiera de *propriétaire* de la variable. La seconde approche est d'avoir plusieurs copies de la variable, à raison d'une copie dans la mémoire de chaque station. Cette deuxième approche soulève cependant le problème de cohérence entre les diverses copies de la variable.

Lorsqu'il n'existe qu'une seule copie d'une variable partagée, on peut distinguer deux types d'accès à cette variable. Lorsque le propriétaire de la donnée accède à celle-ci, soit pour une écriture ou une lecture, il peut le faire directement par un simple accès mémoire, c'est donc un accès *local*. Par contre, toutes les autres stations devront y accéder par l'entremise du réseau car l'accès à cette variable est *non-local*. Ainsi, lors d'une lecture de cette variable, elles devront envoyer un message contenant l'adresse de la variable pour demander au propriétaire de retourner la valeur contenue à cette adresse. Pour ce qui est des écritures, les stations devront également envoyer un message, mais cette fois-ci, il contiendra une adresse et une valeur. À la réception du message, le propriétaire n'a qu'à affecter la valeur reçue à l'adresse reçue. Aucun message de retour n'est requis. Un accès local est évidemment beaucoup plus rapide qu'un accès non-local puisque le premier n'implique aucun accès au réseau.

Avec la deuxième approche, comme il existe une copie de la variable partagée dans la mémoire de chaque processeur, faire des lectures de celle-ci n'implique qu'un simple accès mémoire local, et ceci est valide pour toutes les stations. Avec cette approche, le temps pour une lecture est très petit et en plus il est le même pour toutes les stations. Pour ce qui est de l'écriture avec cette méthode, il faut tout d'abord aborder le problème de cohérence des copies.

Si une station se contente de faire l'écriture désirée uniquement à sa copie locale de la variable partagée, il y a alors un grand risque que les copies ne soient plus *cohérentes* (à

moins que toutes les stations aient par hasard fait la même écriture en même temps...). Avec cette méthode, l'écriture se fait comme la lecture, c'est-à-dire par un simple accès mémoire local et son temps d'exécution est constant pour toutes les stations. Cependant, comme le but de notre mémoire MPV est de permettre à ParSubC d'offrir un modèle de programmation à mémoire partagée, cette approche n'est alors pas adéquate car avec le modèle traditionnel de mémoire, une variable est toujours unique et ne possède donc qu'une seule valeur. Notre MPV a donc l'obligation de tenir les copies d'une variable cohérentes. Ainsi, lorsqu'une station fait une écriture, elle doit propager, à l'aide du réseau, la nouvelle valeur de la variable à toutes les autres stations, et de cette façon, les copies d'une variable globale resteront cohérentes. Cependant, cette méthode pour l'écriture d'une variable partagée fait de l'affectation une opération assez coûteuse puisqu'elle requiert, pour n stations, n-1 envois de messages sur le réseau.

Variables partagées dans ParSubC

Les deux approches pour implanter les variables partagées ont leurs forces et leurs faiblesses, et leur efficacité dépend grandement de l'utilisation que l'on fait des variables partagées. L'implantation de ParSubC utilise donc les deux approches pour implanter les variables globales et les variables locales.

Pour ce qui est des variables locales, il est possible de les partager grâce aux pointeurs, mais ce n'est que très peu fréquemment. Mis-à-part les pointeurs, la portée des variables locales est limitée à la tâche dans laquelle elles sont définies, et donc la majorité des accès à ces variables viennent du propriétaire. Il est donc grandement avantageux de privilégier l'accès local, ce pourquoi dans ParSubC les variables locales sont toujours en copie unique, qu'elles soient partagées ou pas.

Pour les variables globales, c'est plutôt l'autre approche qui a été retenue. Nous avons basé ce choix sur le fait que les variables globales sont plus souvent référencées qu'affectées. Ce choix est cependant beaucoup plus discutable car de cette façon, l'affectation est très coûteuse et certains programmes en souffriront beaucoup. Il reste à évaluer si la très grande rapidité avec laquelle la lecture se fait, comblera la lenteur de l'écriture.

4.5.2 Messages et opérations de la MPV

Il a été vu précédemment que c'est le distributeur de messages qui permet de traiter adéquatement les différents messages qui seront envoyés sur le réseau. Il faut donc définir

des types de message pour chaque opération qu'il est possible de faire avec la MPV. On peut résumer les opérations de la MPV aux trois suivantes:

- Référence – Lorsqu'une station fait référence à un pointeur qui pointe sur une variable non-locale, une requête doit être envoyée au propriétaire de la variable afin que la valeur de la variable puisse être retournée.
- Affectation – Lorsqu'une station fait une affectation par l'entremise d'un pointeur à une variable non-locale, un message pour communiquer la nouvelle valeur au propriétaire doit être envoyé.
- Diffusion – Lorsqu'une station fait une affectation à une variable globale, directement ou par l'entremise d'un pointeur, elle doit communiquer la nouvelle valeur à toutes les autres stations (pour garder la cohérence entre les copies de la variable globale).

Référence

Pour effectuer une référence, deux identificateurs de message doivent être définis, soit un pour l'envoi de la requête et un autre pour la réponse. Une première information nécessaire à envoyer pour une référence non-locale est l'adresse de la variable locale. Lors de la réception d'une telle requête, une station n'a qu'à retourner la valeur de la variable située à l'adresse reçue. Cependant, l'adresse n'est pas suffisante pour identifier une variable, puisque l'information retournée est différente selon que le type de cette variable est, par exemple, un `int` ou un `double`. Ainsi la requête doit également préciser le type de la variable non-locale, et dans notre implantation, cette information est contenue dans l'identificateur de message. Le `msg_id` indiquera donc qu'il s'agit d'une référence et précisera le type de la variable.

Un cas particulier pour la référence surgit cependant avec les structures, à cause de leur taille variable. Ainsi, lorsqu'un identificateur de message indique une référence à une structure, le message contient également la taille de la structure en question. La réponse associée à une telle requête sera constituée de deux messages. Un premier pour identifier que le message envoyé est bien la réponse à la requête d'affectation, et non une autre requête quelconque. Ce message contient également la taille de la structure. Le deuxième message, de taille variable, correspond à la structure elle-même.

Une particularité associée à la référence à une variable non-locale est que c'est la seule opération de la MPV qui requiert une réponse. Cela en fait donc l'opération qui

sera la plus affectée par la vitesse du réseau, car la station doit attendre après la réponse avant de pouvoir reprendre l'exécution de sa tâche, et donc plus lent est le réseau plus longtemps la tâche sera suspendue.

Affectation

L'affectation est une opération à sens-unique, c'est-à-dire qu'elle est constituée uniquement d'une requête, il n'y a pas de réponse requise. Ainsi pour affecter une variable non-locale, une station n'a qu'à envoyer l'adresse de la variable et la nouvelle valeur au propriétaire de la variable. Pour le cas présent, comme pour la référence, le type est également requis, et il est compris dans l'identificateur de message. Avec ces informations, le propriétaire est en mesure d'effectuer l'affectation désirée.

Pour l'affectation d'une structure, qui est un type de taille variable, deux messages sont requis: un premier pour identifier la requête et envoyer l'adresse et la taille de la structure et un deuxième correspondant à la structure elle-même.

L'opération d'affectation est plus rapide que la référence du point de vue de la station accédant à la variable non-locale, car elle peut reprendre l'exécution de sa tâche immédiatement après avoir envoyé le message. Elle n'a pas à attendre une réponse avant de continuer.

Diffusion

La diffusion est une opération très similaire à l'affectation puisque son but est également d'affecter une valeur à une variable dans la mémoire d'une autre station. Une diffusion consiste en fait à faire $n-1$ affectations (où n est le nombre de processeurs), et donc les informations nécessaires et le message utilisé sont les mêmes que pour l'affectation. Une diffusion est implantée comme une boucle d'affectations.

Cependant, une caractéristique propre à la diffusion est que son temps d'exécution est proportionnel au nombre de stations qui participent à l'exécution. Par exemple, la diffusion avec 4 stations est beaucoup plus rapide qu'une effectuée avec 64 stations.

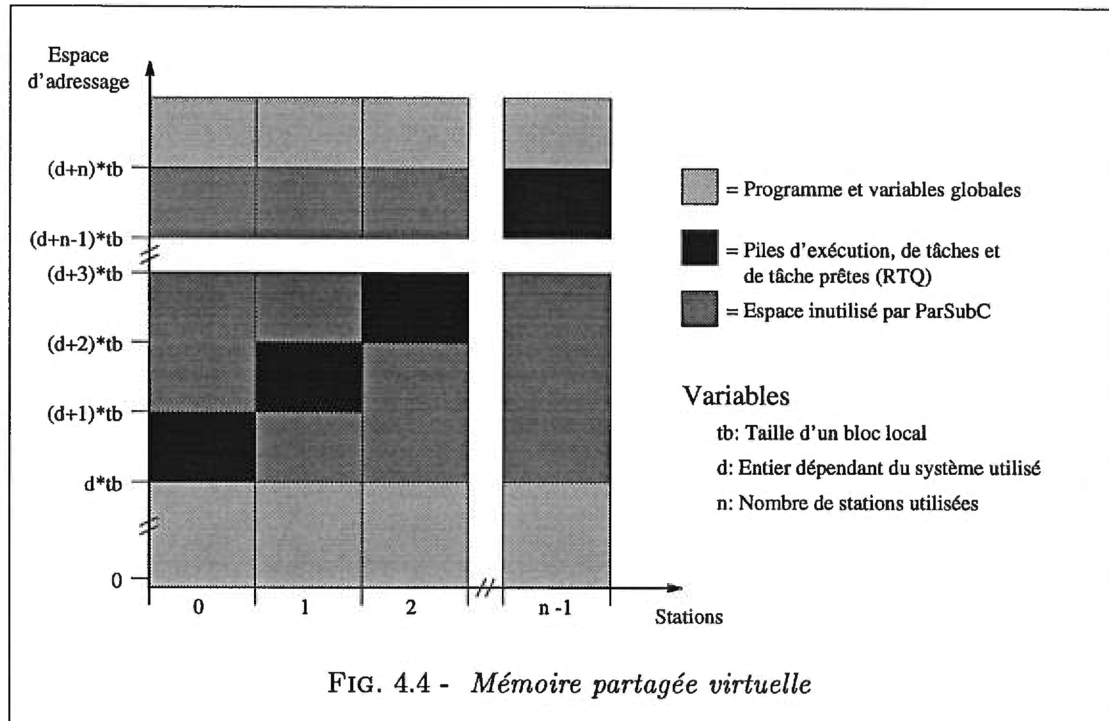

FIG. 4.4 - *Mémoire partagée virtuelle*

4.5.3 Organisation de la mémoire partagée virtuelle

Comme on vient de le voir, la façon dont une variable partagée doit être accédée dépend de son type, soit globale ou locale, et dans le cas d'une variable locale, il faut aussi connaître le propriétaire. Un problème fondamental que doit régler la MPV est donc de déterminer cette information lorsqu'en présence d'une variable partagée. En temps normal, la seule information disponible sur une variable à l'exécution est son adresse. Une première approche serait donc d'ajouter un champ à la variable qui fournirait l'information nécessaire pour y accéder (par exemple -1 si la variable est globale, autrement ce champ contiendrait la valeur identifiant le propriétaire). Cependant, une telle méthode a le désavantage d'ajouter un coût pour gérer ce champ, en plus d'augmenter l'espace pris par les variables partagées.

Une méthode beaucoup plus astucieuse et efficace est d'utiliser l'adresse de la variable pour identifier son type et son propriétaire. Pour cela, il faut réserver un intervalle d'adresses pour les variables globales, ainsi qu'un intervalle pour les variables locales de chaque station. De cette façon, il suffit de déterminer à quel intervalle appartient une variable pour déterminer son type et pour les variables locales, son propriétaire.

La figure 4.4 montre la structure de la mémoire partagée virtuelle de ParSubC.

L'espace mémoire peut se diviser en deux classes, soit la mémoire réservée pour la MPV qui contient les variables locales et le reste qui contiendra le programme à exécuter et les variables globales.

La mémoire réservée est allouée de façon à pouvoir facilement et rapidement déterminer qui est le propriétaire d'une variable située dans cette mémoire. Ainsi les blocs dont elle est constituée, sont alloués de manière contiguë et ont tous la même taille. De plus, l'adresse du début de la mémoire réservée correspond à un multiple de la taille des blocs. En rappelant que les stations sont identifiées par un entier entre 0 et *n-1*, où n est le nombre de stations, cela permet de réduire le calcul[1] du propriétaire d'une variable à:

$$propritaire = (adresse/taille_bloc) - nb_blocs_depart$$

Tout le reste de l'espace d'adressage peut être utilisé de façon normale, on y retrouvera donc le programme ParSubC ainsi que les variables globales. Ainsi, si la valeur retournée par le calcul du propriétaire est négative ou supérieure à $n - 1$, c'est que l'adresse utilisée ne correspond pas à une variable locale mais plutôt à une variable globale. De cette façon, une adresse est suffisante comme information pour déterminer le type et le propriétaire d'une variable partagée.

Il y a cependant une dernière restriction quant à l'organisation de l'espace mémoire pour l'implantation des variables globales. Pour que les stations puissent diffuser la nouvelle valeur d'une variable globale, il leur faut connaître l'emplacement de la variable dans la mémoire des autres stations. Pour régler ce problème, les variables globales seront situées à la même adresse dans la mémoire de chaque station lors du chargement du programme. Ainsi la diffusion consiste à affecter une valeur à la même adresse dans le mémoire de chaque station.

Modification dynamique du nombre de stations

Une note intéressante à propos de l'implantation de la mémoire partagée virtuelle est que c'est une implantation qui rend très complexe la modification du nombre de stations lors de l'exécution parallèle d'un programme, plus particulièrement en ce qui concerne le retrait d'une station. Par exemple, il serait très difficile de retirer une station lors de l'exécution lorsque des données ont été allouées dans sa mémoire. Dans un tel cas, il faudrait voir à relocaliser les données et en assurer la coordination avec les tâches qui pourraient y faire référence, ce qui est une tâche très complexe.

1. La division utilisée est une division entière arrondie au plus grand entier inférieur.

```
void f(light int *l)
{
  heavy int *h;
  int t;

  /* Supposons que l (et h) pointe sur une variable
     non-locale */
  h = l;

  /* Migration de la donnee vers le proprietaire */
  *l = 10;
  ...

  /* Migration de la tache vers le proprietaire */
  *h = 10;
  ...
}
```

FIG. 4.5 - *Exemple d'annotation de pointeurs*

4.5.4 Identification des variables partagées

Nous avons identifié deux types de variable pouvant impliquer l'utilisation du réseau, soit les variables globales et les pointeurs. Pour les variables globales, il faut donc, lors de la génération de code par le compilateur, inclure les communications nécessaires pour diffuser la valeur affectée à une variable de ce type. Cependant, avec les pointeurs, il faut avant d'utiliser le réseau, déterminer quel est le type ou le propriétaire de la donnée pointée, car les actions à prendre seront différentes selon ces informations.

Comme il y a un coût associé au calcul de ces informations, s'il faut faire ce calcul pour tous les pointeurs, ce coût peut réduire de beaucoup les performances. Lorsque l'indirection d'un pointeur mène à l'utilisation du réseau, le coût associé au calcul est négligeable si on le compare au temps pris pour envoyer un message sur le réseau. C'est plutôt lorsqu'un pointeur pointe sur des données locales à la station que ce calcul peut affecter les performances. Le temps pris pour un accès à la mémoire est très petit, et donc avoir à faire un calcul avant chaque accès fait de la référence locale une opération beaucoup plus lente. Ainsi, l'utilisation intense d'un tel pointeur (dans une boucle par exemple) peut avoir un impact sur les performances d'un programme. Il serait donc intéressant de pouvoir identifier les pointeurs qui *risquent* d'utiliser le réseau des autres pointeurs, pour éviter d'effectuer des calculs inutiles.

Il serait possible d'effectuer une analyse pour détecter les pointeurs qui n'accèdent uniquement qu'aux données locales d'une station, mais c'est une analyse qui peut s'avérer assez complexe, et les informations disponibles à la compilation sont souvent insuffi-

santes pour arriver à une analyse précise. Comme cette implantation de ParSubC n'est qu'un prototype, nous ne nous sommes pas attaqués à une telle analyse.

L'approche utilisée par ParSubC est plutôt de laisser au programmeur le soin d'identifier les pointeurs sur des variables partagées. Le programmeur possède plus d'information que le compilateur et par conséquent, peut arriver à une meilleur analyse, et donc obtenir de meilleurs performances. Pour pouvoir identifier ces pointeurs, le programmeur dispose de deux qualificatifs de type, soit `light` et `heavy`, qui lui permettent d'annoter les pointeurs *à risque* lors de leur déclaration. La figure 4.5 illustre des exemples de déclaration avec ces qualificatifs.

Migration de donnée

Le qualificatif `light` est utilisé pour indiquer l'utilisation *légère* d'un pointeur, c'est-à-dire que ce pointeur ne sera pas utilisé de façon intensive. À la compilation, le code généré calculera le type/propriétaire de l'adresse pointée pour déterminer les actions à prendre pour accéder au contenu de cette adresse. Lorsque le réseau doit être utilisé, soit pour envoyer une valeur à affecter à cette adresse, soit pour recevoir la valeur à cette adresse, on dit alors qu'il y a *migration de donnée.* Le qualificatif `light` permet donc la migration de donnée.

Migration de tâche

Lorsqu'un pointeur est utilisé plus intensivement et qu'il ne pointe pas dans la mémoire locale, cela résulte en beaucoup de migrations de donnée. Comme migrer une donnée est une opération beaucoup plus lente qu'un accès à la mémoire locale, il pourrait être préférable que la tâche faisant ces migrations de donnée soit exécutée par le propriétaire de la donnée pointée. De cette façon, il n'y aura plus de migrations de donnée mais de simples et rapides accès locaux. C'est pour permettre cela que ParSubC possède le qualificatif de type `heavy`. Lorsqu'un pointeur est annoté avec ce qualificatif, si la donnée pointée n'est pas dans la mémoire locale de la station, c'est toute la tâche qui est migrée, et non la donnée. Après la *migration de tâche*, les prochaines utilisations de ce pointeur résulteront en des accès à la mémoire locale. Contrairement à la migration de donnée, la migration de tâche n'utilise pas la MPV, mais est plutôt réalisée de façon similaire à un vol de tâche, et relève donc plutôt du partitionnement. Les ajouts faits au partitionnement pour supporter la migration de tâche seront présentés plus loin dans la section 4.6.1.

```
void f(heavy int *h)
{
  light int *ptr;
  int a;

  a = 0;
  ptr = &a;
  ...

  /* Migration de tache */
  *h = 10;
  ...

  /* Est-ce que ptr pointe encore sur a... */
  *ptr = 10 ;
  ...
}
```

FIG. 4.6 - *Migration de tâche: pointeur sur une variable locale*

4.5.5 Migration de tâche et variables locales

Lors de la migration de tâche, toutes les variables locales à la tâche seront migrées avec la tâche. Cela est essentiel pour avoir une exécution efficace, sinon il faudrait passer par le réseau pour accéder aux variables qui ne seraient plus locales à la station exécutant la tâche. Un problème se pose cependant lorsqu'on utilise un pointeur sur les variables locales. Pour illustrer ce problème, regardons l'exemple de la figure 4.6.

Supposons que la fonction `f` est d'abord exécutée sur la station 0 pour être ensuite migrée vers la station 1. Le pointeur `ptr` est initialisé pour pointer sur la variable locale `a`, et pointe donc quelque part dans l'intervalle d'adresse de la station 0. Par la suite, la tâche est migrée et c'est là que le problème survient, car la variable `a` a migré avec la tâche mais `ptr` pointe toujours dans la mémoire de la station 0.

Une première approche pour résoudre ce problème serait de réajuster le pointeur `ptr` après la migration pour qu'il pointe sur la nouvelle valeur. Ainsi l'exécution pourrait continuer comme si la tâche n'avait jamais été migrée. La difficulté majeure avec cette approche est que la valeur du pointeur `ptr` a pu se propager entre son initialisation et la migration de tâche, et il peut donc y avoir d'autres pointeurs nécessitant un réajustement. Déterminer quels pointeurs doivent être réajustés est très difficile surtout avec une exécution parallèle, puisque les autres stations ont peut-être participé à la propagation de l'ancienne adresse de `a`, et il devient alors pratiquement impossible de détecter toutes les propagations.

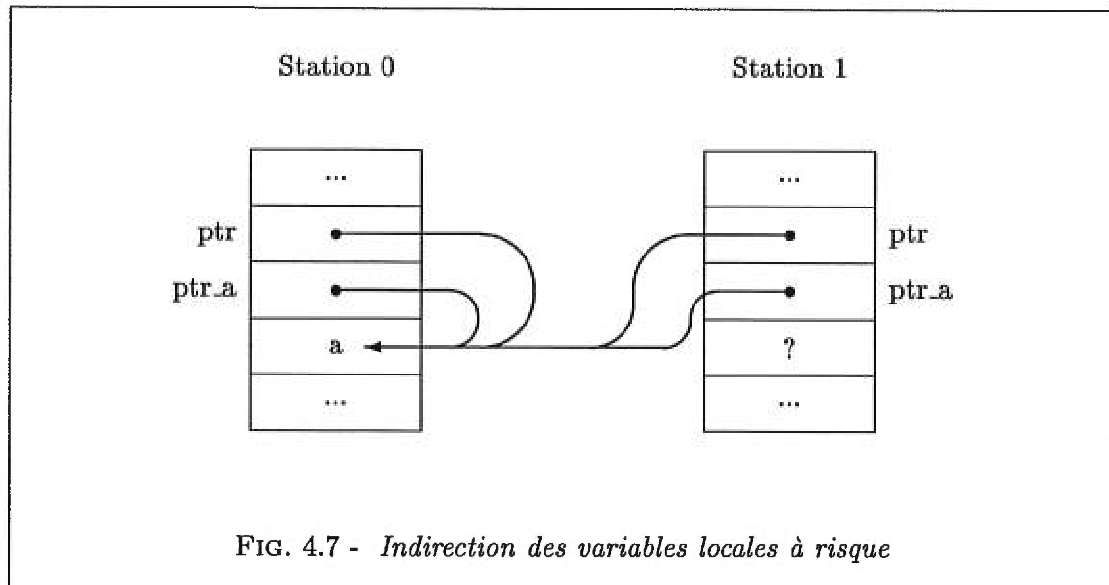

FIG. 4.7 - *Indirection des variables locales à risque*

L'approche adoptée par ParSubC est plutôt de ne pas migrer les variables locales *à risque* (l'expression “à risque” est définie plus loin). De cette façon, il n'est plus nécessaire de réajuster les pointeurs puisque la variable `a` demeure à la même adresse. Cependant, le code généré pour accéder à la variable `a` n'est plus le même car on doit maintenant y accéder comme étant un pointeur sur une donnée, et de plus, il faut traiter ce pointeur comme étant `light` puisqu'après la migration de tâche, il ne pointera plus sur une donnée locale à la station. Cela rend l'utilisation de la variable `a` plus lente, mais le plus important est que l'exécution se fasse correctement et respecte le modèle de programmation à mémoire partagée. De plus, tous les détails sont pris en charge par le compilateur, ce qui facilite le travail du programmeur.

Il y a cependant une lacune avec cette méthode car il se peut que certains pointeurs n'ayant pas à être déclarés `light` sans la migration de tâche, doivent l'être maintenant. C'est le cas du pointeur `ptr` de notre exemple, car après la migration de tâche, il pointe maintenant sur une donnée non-locale à la station. Faire la détection de ces pointeurs lors de la compilation n'est pas réaliste et ce pour les mêmes raisons de propagation de pointeurs évoquées précédemment. Le programmeur a donc la responsabilité d'annoter les pointeurs sur des variables locales qui pourraient devenir non-locales après une migration de tâche.

ParSubC considère qu'une variable locale est *à risque* lorsque son adresse est utilisée

dans une expression. On peut faire référence à l'adresse d'une variable de deux façons:

- L'opérateur `&` lui est appliqué.
- La variable est un tableau, et elle est référencée seule, c'est-à-dire sans l'opérateur `*`, ni `[]`.

Cette définition de variable locale *à risque* est très large, mais elle est simple et, fait encore plus important, elle inclut tous les candidats dangereux.

4.5.6 Cohérence des variables globales

Lorsqu'une station assigne une nouvelle valeur à une variable globale, la diffusion de cette valeur à toutes les stations vise à garder la cohérence entre les multiples copies de la variable. Cependant, cela n'est pas suffisant pour assurer la cohérence des variables globales, car il existe des situations de course où les copies d'une variable peuvent devenir incohérentes. Si deux stations, *S1* et *S2*, font des affectations simultanées à la même variable globale, certaines stations pourraient recevoir le message de *S1* en premier alors que d'autres auront plutôt reçu le message de *S2* en premier. À la fin des affectations, cela résultera en des valeurs différentes de la même variable globale.

Il aurait été possible de modifier la méthode de diffusion des variables globales pour assurer en tout temps la cohérence des variables globales. Pour régler ce problème, il faudrait séquentialiser les affectations, de sorte que les diffusions ne soient pas entre-croisées. Il aurait été possible, par exemple, de n'avoir qu'une seule station qui puisse diffuser les données. Lorsque les autres stations désireraient faire une affectation, elles communiqueraient leurs intentions au *serveur*, et c'est celui-ci qui diffuserait la donnée aux autres stations.

Cependant, cette méthode de diffusion apporte un nouveau problème face à la cohérence des variables globales avec notre MPV. Avec la migration de tâche, il se pourrait qu'une tâche soit exécutée avec des variables globales incohérentes. Supposons que lors de l'exécution d'une tâche, une station assigne une valeur à une variable globale pour ensuite migrer la tâche (à cause d'une référence à un pointeur annoté `heavy`). Si l'exécution de la tâche migrée reprend avant que le serveur ait finit la diffusion de la nouvelle valeur de la variable globale, il y a risque que l'exécution de la tâche migrée se fasse avec l'ancienne valeur pour une certaine période de temps. Pour avoir un modèle de variables globales totalement cohérent, une station qui assigne une nouvelle valeur à une variable

globale devrait donc attendre que le serveur ait terminé la diffusion avant de reprendre l'exécution de sa tâche courrante (par exemple, le serveur pourrait envoyer un message à la station).

Le principal problème que pose cette méthode de diffusion des variables globales est sa lenteur d'exécution. Avec des programmes ayant une fréquence élevée d'affectations à des variables globales, l'utilisation d'un serveur pour effectuer les diffusions peut provoquer un problème de contention, ce qui réduit la performance de la MPV. De plus, effectuer la diffusion requiert maintenant l'intervention de deux stations et ce, pour la durée complète de la diffusion, ce qui fait de la diffusion une opération encore plus coûteuse.

Comme notre MPV repose sur un réseau Ethernet qui est un média très lent par rapport à une vrai mémoire partagée physique, par soucis d'efficacité, nous n'avons pas implanté le concept de serveur. Nous préférons offrir une implantation de ParSubC plus efficace à défaut de ne pas assurer totalement la cohérence des variables globales.

Une première approche pour éviter les problèmes de cohérence avec la programmation avec ParSubC est d'avoir une phase séquentielle pour l'initialisation des variables globales, suivit d'une phase parallèle dans laquelle les variables globales ne sont pas modifiées (ou du moins, les modifications se font de façon séquentielle par les stations).

Si un programme ParSubC ne peut être écrit de façon à garantir la modification séquentielle des variables globales dans sa phase parallèle, une autre approche est alors disponible pour le programmeur. Le problème avec l'implantation des variables globales est qu'il y a plusieurs copies de la même donnée. Pour éviter cela, une autre approche pour partagée une donnée est d'allouer la variable dans la mémoire locale d'une station et d'initialiser un pointeur global à l'adresse de la donnée partagée. De cette façon, toutes les stations ont accès à la variable, et comme il n'y a qu'une seule copie, il ne peut y avoir de problème de cohérence.

4.6 Partitionnement CPT

La CPT utilisée dans l'implantation pour mémoire distribuée est basée sur le protocole à envoi de messages. Ainsi une station de travail ne peut accéder directement aux piles de tâches des autres stations et doit plutôt envoyer des messages à celle-ci. En retour, cette autre station envoie les informations correspondantes à la requête reçue. Pour communiquer à l'aide de messages, nos stations utilisent la composante de com-

```
#define migrate_task(owner, ra)
{
  /* On retarde le traitement des messages. */
  ASYNC_RECV_DISABLE();

  /* On installe la tache dans le pile de taches. */
  _tp->fp = _fp;
  _tp->state = MIGRATED;

  /* On migre la tache vers le proprietaire. */
  SEND_REQUEST(owner, TASK_MIGRATION_MSG, _tp++, NULL, ra));
  SEND_DATA(owner, _fp, _control_points[ra][1]*sizeof(_word));

  /* On permet le traitement des messages. */
  ASYNC_RECV_ENABLE();
  goto _stolen;
}
```

FIG. 4.8 - *Code de migration de tâche*

munication présentée précédemment. Ainsi, lorsqu'une station recherche du travail, elle envoie à une autre station une requête de vol. Celle-ci peut alors lui retourner un des deux messages suivants: soit un message d'échec lui indiquant qu'elle ne possède pas de tâches disponibles, ou soit un message de succès. Dans le cas d'un succès, le message est succédé par les informations concernant la tâche (soit une adresse de retour et un identificateur de tâche (`tid`)). Lorsque le voleur a terminé l'exécution de la tâche volée, il doit alors l'indiquer à la victime et retourner le résultat, s'il y en a un. Pour cela, il envoie un message correspondant au retour de vol, suivi du `tid` et possiblement d'un résultat. Le `tid` est nécessaire car il permet à la victime de savoir de quelle tâche il s'agit et elle peut alors changer l'état de la tâche dans la pile de tâches.

La seule différence qu'on retrouve entre la CPT présentée dans le chapitre 3 et celle de notre implantation, est la présence de la migration de tâche introduite par notre mémoire partagée virtuelle à l'aide de l'annotation `heavy`.

4.6.1 Migration de tâche

Il peut être nuisible pour les performances d'avoir une tâche qui utilise intensément une variable qui n'est pas dans la mémoire de la station qui l'exécute, puisque le temps d'accès pour notre MPV est beaucoup plus élevé que celui d'un accès à une mémoire locale. C'est dans ce cas qu'il devient préférable d'utiliser l'annotation `heavy`, car cela permet de migrer la tâche vers la station qui possède la donnée. L'opération de migration de tâche est très semblable à celle d'un vol de tâche, car il s'agit d'envoyer une tâche

pour qu'elle soit exécutée par une autre station. La seule distinction est que la station qui reçoit la tâche migrée n'en a jamais fait la demande. Malgré cela, nous appellerons également *victime* le processeur d'où origine la tâche, et *voleur* le processeur qui reçoit la tâche migrée.

La figure 4.8 montre le code associé à la migration de tâche. On peut voir que la migration consiste simplement à empiler une tâche légère dans la pile de tâches pour ensuite l'envoyer au propriétaire de la variable qui a provoqué la MT. L'état de cette tâche est alors marqué comme étant migrée. A partir de ce moment, la tâche migrée est traitée comme une autre tâche volée. Pour cela, la tâche migrée possède un espace dans le tableau des points de contrôle qui contient les mêmes informations qu'une tâche volée, soit le nombre d'arguments, la taille du résultat ainsi que le point d'entrée. Comme une tâche migrée ne correspond pas à un appel de fonction (comme c'est le cas pour une tâche volée), le nombre d'arguments ne désigne pas le nombre de paramètres, mais plutôt le nombre de variables locales qui figuraient sur la pile d'exécution au moment de la migration. Pour ce qui est du point d'entrée, il correspond à l'endroit qui a déclenché la MT, soit l'accès au pointeur `heavy` pointant sur une donnée non-locale. Lors de la génération du code, il faut donc ajouter des étiquettes à toutes les indirections de pointeur `heavy`, car ce sont tous des points d'entrée potentiels.

Du point de vue de la station recevant une tâche migrée, il n'y a aucune différence avec la réception d'une tâche volée. Elle ne sait d'ailleurs pas si la tâche reçue est migrée ou volée. Tout ce dont elle s'aperçoit est qu'il y a une tâche prête à être exécutée dans sa pile de tâches prêtes. Elle l'exécutera donc normalement pour ensuite retourner le résultat à ce qu'elle perçoit comme étant une victime.

Une distinction importante à faire en ce qui concerne la tâche migrée, est que ce n'est pas vraiment la tâche qui a rencontré le point de migration qui est migrée, mais plutôt le reste de la fonction contenant ce point. Cela ne fait pas une grande différence lorsque la tâche correspond uniquement à la fonction, mais dans le cas où la tâche correspond à plusieurs appels de fonction imbriqués, c'est seulement le dernier appel de fonction qui est migré. Lorsqu'un processeur désire migrer une tâche, il empile donc une tâche correspondant au reste de la fonction dans sa pile de tâches et c'est cette tâche qu'il envoie au propriétaire. Ainsi le travail du processeur qui reçoit la tâche migrée consiste à terminer l'exécution de la fonction qui a causé cette migration. La raison pour migrer le dernier appel de fonction uniquement est que cela simplifie grandement l'implantation, en plus de la rendre plus efficace.

La difficulté reliée à la migration d'une tâche complète est que le processeur qui exécute la tâche avant sa migration peut avoir empilé des tâches légères. La figure 4.9

```
void f1 (heavy int *t)
{
   ...
   *t = 10;
   ...
}

void f2 (heavy int *t)
{
   ...
   *t = 10;
   ...
}
```

```
void g (heavy int *t)
{
   ...
   h() !!
   {
      f1(t);
      f2(t);
      *t = 10;
   }
   ...
}
```

FIG. 4.9 - *Tâche légère associée à une tâche à migrer.*

montre un exemple de programme pouvant créer une telle situation. Dans cette figure, on suppose que la fonction g correspond à la tâche en exécution. Au cours de l'exécution de la fonction g, une tâche légère associée à la fonction h est empilée. Dans le bloc d'énoncé associé à cette tâche se trouve un appel à la fonction `f1`. Si l'exécution de la fonction `f1` cause une migration de tâche, alors il y donc une tâche légère h associée à la tâche à migrer. Il faudrait donc voir à migrer la tâche légère également. Il y a trois principaux problèmes découlant de la migration de la tâche associée à une tâche à migrer:

- Dans cet exemple, il n'y a qu'une seule tâche légère associée à la tâche à migrer, mais avec la récursivité, ce nombre peut prendre n'importe quelle valeur. Pour effectuer la migration de tâche, il faut donc déterminer combien de tâches légères doivent être migrées.
- Les champs `fp` des piles légères pointent sur la pile d'exécution du propriétaire de la tâche à migrer, et non celle du destinataire.
- Si certaines des tâches légères ont déjà été volées, le résultat sera retourné au propriétaire de la tâche à migrer et non pas celui qui exécute la tâche migrée.

Du point de vue de l'efficacité de l'implantation, migrer uniquement le dernier appel de fonction est plus efficace, car il ne requiert pas tout le travail ci-haut. De plus, migrer la tâche complète serait moins efficace au niveau réseau, car la tâche pourrait être très grosse puisqu'elle pourrait contenir plusieurs appels de fonction imbriqués et, en plus, il faudrait migrer toutes les tâches légères s'y rattachant.

Cependant, il existe certains programmes qui seraient avantagés, au point de vue efficacité, par la migration de la tâche au complet. C'est le cas de l'exemple de la figure 4.9, si lors de l'exécution de la fonction `g`, l'exécution de `f1` et de `f2` causent toutes deux des migrations de tâches vers la même stations. Dans un tel cas, il aurait été préférable de migrer la tâche `g` au complet, ainsi il n'y aurait eu qu'une seule migration de tâche. Pour certains problèmes, la migration de la tâche complète peut donc éviter plusieurs migrations de tâches.

Migration de tâche et opérateur `!!`

Tel qu'il vient d'être vu, il peut être très problématique qu'il y ait une tâche légère associée à la tâche à migrer. C'est pourtant le cas lorsque le point de migration est à l'intérieur d'un bloc d'énoncés associé à l'opérateur `!!`. Un exemple de programme qui peut causer une telle situation est l'énoncé `*t = 10` de la fonction `g` de la figure 4.9. Pour contourner ce problème, notre implantation de ParSubC n'effectue pas de migration de tâche aux points de migration contenus dans de tels blocs d'énoncés, mais se contente plutôt de faire une simple migration de donnée. Permettre la migration de tâche uniquement à l'extérieur des blocs d'énoncés associés à l'opérateur `!!` a simplifié grandement l'implantation de la migration de tâche.

Migrations subséquentes d'une tâche

Lorsqu'une tâche est volée, celle-ci a d'abord du être empilée dans la pile de tâches de la victime. Si, lors de l'exécution de la tâche par le voleur, cette même tâche doit être migrée vers une autre station, elle doit avant sa migration, être empilée dans la pile de tâches du voleur et cette tâche se retrouve donc dans deux piles de tâches différentes. Pour chaque migration subséquente de la tâche, celle-ci occupera un nouvel emplacement dans une pile de tâches. Avec la MT, une même tâche peut ainsi occuper plusieurs piles de tâches et peut même se retrouver plusieurs fois dans une même pile.

4.6.2 Ordonnancement et migration de tâche

La migration de tâche est une opération qui s'effectue de façon très semblable au vol de tâche. On met donc la tâche dans la pile de tâche et on l'envoie à la station qui possède la variable pointée par le pointeur annoté `heavy`. La seule différence par rapport au vol de tâche, est que dans le cas d'une tâche migrée, il n'y a jamais eu de

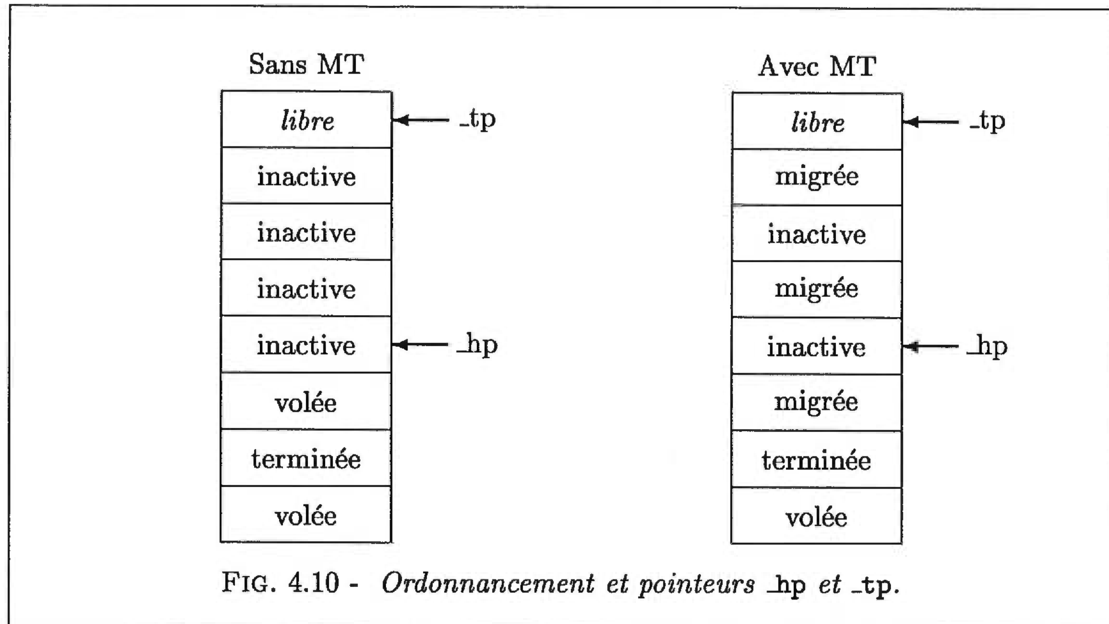

FIG. 4.10 - *Ordonnancement et pointeurs* `_hp` *et* `_tp`.

demande explicite pour celle-ci. Cependant, malgré que la MT ne semble pas demander beaucoup de changements, certains sont requis car sa présence modifie l'ordonnancement des tâches.

Sans la MT, un processeur peut facilement déterminer s'il possède une tâche disponible dans sa pile ; il lui suffit de vérifier si son pointeur du bas de pile de tâches (`_hp`) est égal à celui du haut de la pile (`_tp`). Pour comprendre pourquoi cela est valide, il faut se rappeler que les tâches sont toujours volées par le dessous alors que le propriétaire lui se sert par le dessus. Une autre propriété intéressante de la pile de tâches est que lorsque les deux pointeurs sont différents, les tâches comprises entre elles sont toujours disponibles. Par exemple, dans la pile de tâches de gauche de la figure 4.10, deux tâches sont disponibles. Ainsi il est facile de savoir s'il y a des tâches disponibles dans la pile de tâches et il est aussi simple d'en extraire une lorsqu'il y en a, puisque `_hp` pointe sur la prochaine tâche à voler et `_tp` pointe au-dessus de la prochaine à dépiler.

Avec la MT, lorsque les pointeurs sont égaux, il n'y a pas de tâches disponibles, cependant l'inverse n'est plus toujours vrai. Il se peut que les pointeurs soient différents mais que la ou les tâches contenues entre elles aient été migrées. Ainsi lorsqu'une station cherche une tâche disponible, il arrive qu'elle ait à parcourir sa pile de tâches puisqu'elle doit passer par dessus toutes les tâches ayant été migrées (voir la pile de droite de la figure 4.10). Pour pouvoir distinguer les tâches volées des tâches migrées, un nouvel état, soit *migrée*, a été ajouté. Quoique cela ne soit pas très compliqué, il n'est donc quand

même plus aussi simple de déterminer s'il y a des tâches disponibles et d'en trouver.

Dans le cas d'un vol de tâche, cela n'affecte pas vraiment les performances car parcourir une pile est une opération beaucoup plus rapide que d'envoyer des messages à travers le réseau. Ainsi les communications restent toujours le facteur ayant le plus d'impact sur les temps d'exécution. Pour ce qui est du dépilement des tâches par son propriétaire, la MT introduit un nouveau scénario face à la suspension tâche. Comme il a été vu dans le chapitre précédent, une station doit suspendre l'exécution de sa tâche courante lorsque la tâche sur le dessus de sa pile de tâches a été volée et que le voleur n'a pas terminé de l'exécuter. Sans la MT, le fait que la tâche du dessus ait été volée impliquait que toutes les tâches avaient été volées et, par conséquent, il ne servait à rien pour une station de fouiller dans sa propre pile de tâches. Avec la MT, lorsqu'une tâche est migrée, cette tâche correspond à la tâche du dessus de la pile et il doit donc automatiquement y avoir une suspension de tâche. Cependant, comme on peut le voir avec la pile de droite de la figure 4.10, il n'est plus vrai que la suspension de tâche implique qu'il n'y ait plus du tout de tâches dans la pile, et par conséquent, la pile de tâches constitue donc une source potentielle de travail qui mérite d'être fouillée avant d'essayer de voler une tâche à une autre station.

La fonction `get_work` présentée dans le chapitre précédent doit ainsi être modifiée pour inclure cette nouvelle source de travail (la figure 4.11 montre le nouveau cas à considérer lors de la recherche de travail). Malgré cela, l'impact que ces nouvelles suspensions de tâche peuvent avoir sur les performances ne constitue pas vraiment un problème si l'on considère que cela permet d'éviter de nombreuses communications (ce qui est le but de la migration de tâche).

Nécessité du RTQ

Il a été présenté dans le chapitre sur la CPT une pile servant à recevoir les tâches volées. Cette pile, nommée pile de tâches prêtes (RTQ), permet lors de la réception d'une tâche, suite à une requête de vol, de conserver la tâche jusqu'à ce qu'elle soit installée dans la pile d'exécution. Jusqu'à présent, la présence d'une telle pile n'était pas justifiée car il ne pouvait y avoir qu'une seule tâche à la fois dans la pile de tâches prêtes puisque les stations ne demandent du travail que lorsqu'elles n'en ont plus et qu'elles le font en envoyant qu'une seule requête à la fois. Une structure pouvant contenir une seule tâche semble donc suffisante. Cependant, avec la présence de la MT, le RTQ est maintenant nécessaire car une station peut maintenant recevoir des tâches prêtes pour l'exécution, sans en avoir fait explicitement la demande. Ainsi, il peut donc y avoir plusieurs tâches

```
/* Si la derniere tache a ete migree, il y a peut-etre des taches
   disponibles dans la pile. */
if ( ((_gtp-1)->state == MIGRATED) )
{
  _task *tid = _gtp-1;

  /* On parcours la pile. */
  while (tid->state != INACTIVE) tid--;

  /* Si on n'est pas en dessous de _hp, c'est que tid pointe sur
     une tache inactive. */
  if (tid >= _hp)
  {
    _word *stolen_fp;
    int ra, pc, n1, n2, i;

    /* On doit conserver les arguments de la tache migree. */
    _gfp += _control_points[*(int*)(_gfp-1)][1];

    /* Informations necessaires a la creation de tache. */
    tid->state == STOLEN;
    stolen_fp = tid->fp;
    ra = *(int *)((stolen_fp-1)); /* Adresse de retour */
    pc = _control_points[ra][0];  /* Point d'entree */
    n1 = _control_points[ra][1];  /* Nombre d'arguments */
    n2 = _control_points[ra][2];  /* Taille du resultat */

    /* Installer la tache dans la pile d'execution. */
    *(_task **)(_gfp++) = tid;
    _gfp += n2;
    *(int *)(_gfp++) = 0;
    for (i=0; i<n1; i++) _gfp[i] = *stolen_fp++;

    return pc; /* On retourne le point d'entree. */
  }
}
```

FIG. 4.11 - *Nouvelle source de travail pour la fonction* get_work

migrées à la fois, en plus de la tâche volée. Une station doit pouvoir conserver ces tâches jusqu'à ce qu'elle soit prête à les exécuter, d'où la nécessité du RTQ.

Étreinte fatale ("deadlock") avec la migration de tâche

Avec les changements qu'entraîne la migration de tâche sur l'ordonnancement des tâches dans la pile de tâches et avec la possibilité maintenant d'avoir plusieurs tâches dans le RTQ, il est naturel de se demander si la migration de tâche ne pourrait pas

engendrer des situations d'*étreinte fatale*[2]. Cependant, si on analyse de plus près le fonctionnement de la CPT ainsi que celui de la migration de tâche, on arrive à démontrer que ce n'est pas possible. Pour comprendre cela, on peut se baser sur les deux aspects suivants de la CPT:

- Une station fait l'une de deux choses: soit elle exécute une tâche, soit elle cherche du travail. Une station n'est jamais à attendre passivement qu'une autre tâche se termine.
- Lorsqu'une station cherche du travail, elle fouillera complètement sa pile de tâches ainsi que son RTQ. Ainsi, si elle possède une tâche exécutable, elle la trouvera.

Pour qu'il y ait une situation d'étreinte fatale, cela implique qu'à un moment dans l'exécution d'un programme, toutes les stations sont sans travail. Cela implique donc également que toutes les tâches existentes sont suspendues et, d'après les énoncés ci-dessus, qu'aucune station ne possède de tâches exécutables. Ce sont ces deux dernières affirmations qui sont impossibles et ce, pour la raison qui suit. Il existe deux événements conduisant à la suspension d'une tâche: le vol de tâche (lorsque l'exécution de la tâche volée n'est pas terminée) et la migration d'une tâche. Dans les deux cas, il est important de noter que pour chaque suspension, il existe une tâche exécutable s'y rattachant. Ceci implique donc qu'il existe toujours au moins une tâche exécutable à tout moment de l'exécution d'un programme ParSubC et, par conséquent, qu'il ne peut y avoir de situation d'étreinte fatale même avec la migration de tâche.

4.7 Résumé

L'implantation de ParSubC pour le réseau de stations de travail repose beaucoup sur l'utilisation du réseau puisque les communications inter-processeurs sont essentielles tant pour le partitionnement de tâches que pour offrir un modèle de programmation à mémoire partagée. La CPT utilise donc le réseau pour distribuer les tâches entre les stations et la MPV permet la migration de donnée entre les stations. La communication par envoi de messages à travers le réseau étant un moyen de communication lent, la MPV avec l'aide de la CPT permet également la migration de tâche. De cette façon, certains accès intenses à des variables non-locales, annotées `heavy`, causeront la migration de

2. Pour ceux qui ne seraient pas familier avec cette expression, *étreinte fatale* est la traduction française de l'expression anglaise "deadlock".

la tâche plutôt qu'une grande quantité de migrations de donnée qui pourraient nuire considérablement aux performances.

Chapitre 5

Implantation avec mémoire partagée

Ce chapitre présente l'implantation du langage ParSubC pour une architecture parallèle à mémoire partagée. C'est une implantation qui est plus simple et plus efficace que celle pour mémoire distribuée, principalement à cause de la présence de la mémoire partagée qui permet aux processeurs de communiquer entre eux par simple écriture et lecture en mémoire. C'est là un moyen de communication inter-processeur simple et très rapide qui permet d'implanter efficacement la création paresseuse de tâche.

5.1 Mémoire partagée du CRAY T3D

Malgré que le CRAY T3D possède une mémoire partagée, il y a une ressemblance entre l'architecture de notre multi-ordinateur à mémoire distribuée et le CRAY T3D. Dans les deux cas, l'architecture est composée de processeurs ayant chacun leur mémoire locale et qui sont tous reliés par un réseau. C'est ce réseau qui permet aux deux architectures de se munir d'une mémoire partagée.

Cependant, le CRAY T3D possède certains avantages importants qui permettent de le distinguer du réseau de stations HP. Premièrement, le réseau d'interconnexion du CRAY est beaucoup plus rapide que le réseau qui relie les HP. Les processeurs peuvent donc effectuer des accès aux mémoires non-locales dans des temps se rapprochant beaucoup des temps d'accès normaux de mémoires. De plus, le CRAY offre aux processeurs un mécanisme d'accès aux mémoires non-locales strictement matériel et qui ne nécessite

```
typedef struct
{
  _word *stk, *fp;           /* Pile d'execution et son pointeur */
  _task *ltq, *tp, *hp;      /* Pile de taches et ses pointeurs */
  _word *rtq, *rtt, *rtp;    /* Pile de taches pretes et ses pointeurs */
  int pending_steal_id;      /* -1 ou processeur a qui une requete de vol a
                                ete envoyee */
  int victim_id;             /* Processeur qui a ete la derniere victime */
  int thief_id;              /* -1 ou processeur desirant voler une tache */
} _processor;

_processor _self_processor;
_processor *_processor_table[NB_NODES];
```

FIG. 5.1 - *Structure d'un processeur*

pas une intervention logicielle. Ces deux caractéristiques permettent au CRAY T3D de faire partie de la classe des ordinateurs à mémoire partagée.

5.1.1 Accès à la mémoire partagée

Le CRAY T3D permet à un processeur d'accéder à la mémoire partagée des autres processeurs sans avoir à en faire la demande explicitement comme c'était le cas pour le réseau de stations de travail. Le CRAY T3D peut être configuré de façon à ce que certains bits de l'adresse d'une donnée indiquent quel processeur possède cette variable. Pour être plus précis, ce sont les bits 32 à 35 d'une adresse de 64 bits, qui indiquent de quel processeur il s'agit [Num94].

Comme les variables globales sont toutes contenues aux mêmes adresses d'un processeur à l'autre, un processeur peut, par exemple, accéder au contenu du pointeur `tp` du processeur 4 simplement en prenant l'adresse de son propre pointeur `tp`, et en réglant les bits 32 à 35 de sorte qu'ils indiquent 4. Une référence à cette adresse retournera le contenu du pointeur `tp` du processeur 4.

Variables de la CPT

Par souci d'efficacité et de simplicité, plutôt que d'ajuster les bits d'adresse à chaque fois qu'un processeur veut accéder aux données de la CPT d'un autre processeur, le calcul sera conservé en mémoire. La figure 5.1 montre la structure qui contient les données de la CPT. Chaque processeur possède sa propre copie de cette structure, soit

la variable `_self_processor`, pour ses données locales. Pour accéder aux structures des autres processeurs, une table de pointeurs sur les structures sera initialisée de sorte qu'il y ait une entrée qui pointe vers la copie de la variable `_self_processor` de tous les processeurs. Ainsi, une simple indirection avec la bonne entrée de cette table permet d'accéder aux pointeurs voulus dans la mémoire locale voulue.

5.1.2 Variables globales

Étant donné que chaque processeur possède sa mémoire locale et qu'une copie du programme est démarrée sur chacun des processeurs, toutes les variables globales d'un programme se retrouvent en plusieurs copies. Ceci est désirable pour les variables utilisées pour la CPT (telles `fp`, `tp`, ...), puisqu'elles ont une valeur différente pour chaque processeur. Mais pour les variables globales du programme ParSubC, cela pose le même problème qu'avec l'implantation avec mémoire distribuée: comment garder la cohérence entre ces différentes variables globales?

La solution de la précédente implantation qui consiste à faire des diffusions à chaque fois qu'une variable globale est mise à jour n'est pas très attrayante pour cette implantation. Le problème vient du coût qui sera associé à l'opération de diffusion, qui ne serait plus strictement une opération matérielle puisqu'une boucle serait nécessaire pour mettre à jour toutes les mémoires locales. En contrepartie, garder une seule copie des variables globales serait beaucoup plus naturel et simple puisque les processeurs peuvent facilement accéder à la mémoire d'un autre processeur.

Pour cela, les adresses des variables globales seront initialisées de sorte qu'elles pointent toutes à la mémoire du même processeur (qui sera le processeur 0). Par exemple, plutôt que d'accéder directement à la variable globale `g`, un processeur utilisera à la place un pointeur contenant l'adresse de la variable globale et dont les bits 32 à 35 auront été réglés de manière à pointer vers la mémoire du processeur 0. Avec cette technique, une seule copie des variables est utilisée, ce qui permet d'éviter les problèmes de diffusion.

Pour éviter d'avoir un pointeur pour chaque variable globale, une structure contenant toutes les variables globales est plutôt utilisée. La figure 5.2 montre un exemple de déclaration de variables globales avec cette méthode. Les processeurs possèdent donc chacun un pointeur vers la structure contenant les variables globales. L'espace pour les variables globales est alloué uniquement dans la mémoire du processeur 0 et tous les pointeurs pointent vers cet espace.

```
struct struct_gv
{
  int g;
  char c;
};

/* On accede a la structure par un pointeur */
struct struct_gv *gv;
```

FIG. 5.2 - *Structure de variables globales*

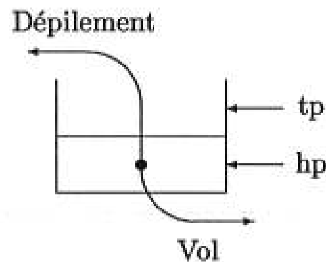

FIG. 5.3 - *Vol de tâche et dépilement simultané d'une tâche (PMP)*

5.2 Partitionnement

Dans les deux chapitres précédents, le modèle de CPT présenté utilise le protocole à envoi de messages. Comme cette technique ne nécessite pas de mémoire partagée, il est donc naturel de l'avoir utilisée pour l'implantation avec mémoire distribuée. Cependant, comme le multi-ordinateur dispose d'une mémoire partagée, il serait intéressant de se demander si le protocole à mémoire partagée ne pourrait pas être plus efficace pour cette implantation.

5.2.1 Protocole à mémoire partagée *vs* à envoi de messages

Avec un protocole à mémoire partagée (PMP), c'est le voleur qui a toute la responsabilité du vol de tâche. Il doit lui-même extraire une tâche de la pile de la victime et la créer pour pouvoir ensuite l'exécuter. L'avantage de cette technique est que la victime n'interagit pas avec le voleur, et elle n'est donc pas dérangée lors des vols de tâche.

```
void f(int i)
{
  ...
  f(i-1) !! { ... };
  ...
}
```

a) Récursivité linéaire

```
int pfib(int i)
{
  int t1, t2;

  if (i<2) return i;
  t1=pfib(i-1) !! {t2=pfib(i-2);};
  return t1+t2;
}
```

b) Doublement récursif

FIG. 5.4 - *Partitionnement et récursivité*

À première vue, ce protocole semble plus performant que le protocole à envoi de messages (PEM), mais il ne faut pas conclure trop rapidement. Le problème avec le PMP est qu'il y a maintenant plusieurs processeurs qui peuvent accéder aux piles de tâche, et il faut donc porter une attention particulière aux accès faits aux piles. Par exemple, si un processeur possède une seule tâche dans sa pile et qu'il essaie de la dépiler en même temps qu'un processeur lui vole une tâche, ces deux processeurs auront pris la même tâche (voir figure 5.3). Pour régler ce problème, il faut donc séquentialiser les accès de sorte qu'il n'y ait qu'un seul processeur qui retire une tâche de la pile à la fois. En d'autres mots, le vol et le dépilement de tâche sont des sections critiques qui doivent être séquentialisées.

Ajouter des opérations nécessaires à implanter les sections critiques pour les vols de tâche n'est pas très grave, car le vol de tâche se fera quand même plus rapidement que s'il fallait interagir avec la victime (comme c'est le cas avec le PEM). Envoyer un message et attendre la réponse de la victime (PEM) sont des opérations plus longues que dépiler une tâche (PMP) même si le dépilement se fait dans une section critique. Là où l'ajout des opérations pour séquentialiser les accès à la pile est ennuyant, est pour le cas du dépilement d'une tâche par son propriétaire qui se voit ralenti par ces opérations. Les deux protocoles ont leur force et leur faiblesse, alors lequel choisir? Avec PEM, le dépilement d'une tâche se fait plus rapidement qu'avec PMP car il ne se fait pas dans une section critique. Il est possible d'implanter des sections critiques de façon efficace, mais il demeure quand même que pour les problèmes où le nombre de dépilements de tâche est très élevé par rapport au nombre de vol de tâche, le PEM est préférable.

Un exemple d'un tel problème est `pfib.c` présenté à la figure 5.4 b. Comme ce programme est doublement récursif, beaucoup de tâches sont générées, et donc un processeur empile (et dépile) beaucoup de tâches. Rapidement après le début de l'exécution du programme de la figure 5.4 b, tous les processeurs auront un nombre élevé de tâches

légères dans leurs piles et ainsi lorsqu'un processeur sera prêt à voler une autre tâche, il aura déjà empilé et dépilé plusieurs tâches légères dans sa pile de tâches. Chaque tâche volée cause donc plusieurs empilements et dépilements de tâche légère. Même si vers la fin du programme ce nombre diminue, il n'en demeure pas moins que le rapport entre le nombre de tâches dépilées et le nombre de vols de tâche reste quand même grand, ce pourquoi il serait préférable, pour de tels programmes, de minimiser le temps pris par le dépilement.

D'un autre côté, comme avec PMP le vol de tâche est plus efficace, les problèmes où le nombre de vols tâche s'approche du nombre de tâches empilées seront plus avantagés par ce dernier protocole. Des exemples de ce style de problème sont ceux à récursivité linéaire (voir la figure 5.4 a), car un processeur ne peut alors jamais avoir plus d'une tâche disponible dans sa pile, et devra donc à chaque fois qu'il se fait voler sa tâche, en voler une lui aussi par après. Durant l'exécution d'un programme à récursivité linéaire, il n'y a jamais plus d'une tâche légère disponible à la fois pour tous les processeurs, ce qui augmente les chances d'une tâche légère de se faire voler. Le nombre de vols de tâche sera donc beaucoup plus près du nombre de dépilements de tâche qu'avec les programmes doublement récursifs.

Aucun des protocoles n'est donc parfait puisque leur efficacité dépend des programmes exécutés. Il ne faut cependant pas perdre de vue, qu'il n'y aura une différence que pour les problèmes dont la granularité du parallélisme est très fine, et que pour les autres problèmes de granularité plus grossière, la différence ne sera pas remarquable. Le protocole qui a été retenu pour notre implantation est PEM parce que c'est également le protocole qui sera utilisé dans l'implantation avec mémoire distribuée (puisqu'il n'y a pas de mémoire partagée), ce qui permet donc de garder plus de similitude entre les deux implantations. Le protocole à mémoire partagée n'offre rien de clairement supérieur au protocole à envoi de messages qui motiverait de changer de protocole.

5.2.2 Techniques d'interruptions pour le PEM

Un mécanisme nécessaire à l'implantation du PEM est celui qui permet aux processeurs de s'envoyer des messages entre eux. Un processeur peut facilement et rapidement envoyer un message à un autre processeur grâce à la mémoire partagée. Cependant, bien que ce deuxième processeur ait reçu rapidement le message, il doit en être averti pour pouvoir le traiter. Minimiser le délai entre la réception du message et son traitement permet de réduire le temps de réponse des vols de tâche. Donc plus petit est ce délai, plus performante sera notre implantation et par le fait même, plus fine sera la

granularité qu'il sera possible d'exploiter.

Dans l'implantation pour mémoire distribuée, la technique d'interruption utilisée pour implanter le PEM est par interruptions matérielles. Ce choix était adéquat car la lenteur du déclanchement des interruptions matérielles n'était pas très déterminante puisque l'envoi de messages à travers le réseau est encore plus lent. Cependant, la mémoire partagée dont est muni le CRAY T3D offre un moyen de communication beaucoup plus rapide et l'impact de la lenteur des interruptions pourrait se faire sentir davantage. Il vaut donc la peine de s'y attarder un peu plus.

Interruptions matérielles

Le principal inconvénient des interruptions matérielles est que c'est un mécanisme lent à déclencher. Pour qu'une interruption puisse être déclenchée, il faut premièrement qu'un signal soit envoyé au processeur, puis l'état de l'exécution courante du processeur interrompu doit être sauvé (changement de contexte), pour ensuite pouvoir brancher à la routine d'interruption, qui traitera la requête. Le temps entre l'envoi de la requête et de son traitement par la routine est donc assez élevé (par rapport au "polling" qu'on verra plus tard).

Cette technique possède cependant deux avantages intéressants. Premièrement, le seul travail fait est lorsqu'il y a une interruption requise par un autre processeur, il n'y a aucun coût associé à quelque vérification que ce soit. Il y aura autant d'interruptions que demandés mais pas plus, et c'est le seul travail nécessaire pour utiliser ce mécanisme (mis à part une initialisation au début du programme).

Le deuxième avantage de cette technique est que, mis à part le temps de déclenchement de l'interruption, il n'y a aucun délai avant le traitement de l'interruption. Dès qu'un processeur envoie une requête d'interruption, elle est immédiatement déclenchée. L'interruption matérielle est donc un mécanisme lent à déclencher mais qui se déclenche uniquement et immédiatement lorsqu'un processeur envoie une requête.

Technique de "polling"

Avec la technique de "polling", un processeur indique à un autre qu'il veut l'interrompre en lui envoyant un message (dans notre cas, il s'agit d'une écriture en mémoire partagée). Lorsque cet autre processeur s'aperçoit qu'il a reçu un message, il traite alors la demande en fonction du message reçu. Un processeur ne reçoit aucun signal lorsqu'il

reçoit un message, d'où la nécessité d'effectuer des vérifications périodiques. Contrairement à la technique précédente, le "polling" est uniquement basé sur des principes logiciels, ce qui rend le déclenchement de ce mécanisme plus rapide.

Le problème cependant avec cette technique est de déterminer à quel intervalle un processeur doit vérifier s'il a reçu une requête d'interruption. Malgré que cet intervalle est ajustable, il n'y a pas une valeur parfaite pour tous les problèmes. Ainsi, si l'intervalle est trop court, un processeur effectuera souvent des vérifications sans qu'il n'y ait de requête et passera donc une partie de son temps à faire des vérifications inutiles. Augmenter l'intervalle réduit le nombre de vérifications, mais augmente également le temps de réponse des requêtes d'interruption, ce qui peut résulter en une diminution des performances. Trouver le juste milieu est une tâche difficile et en plus, il diffère d'un programme à l'autre.

Les deux mécanismes ont leurs avantages et leurs inconvénients, mais il existe une autre distinction et c'est ce qui justifie notre choix lors de l'implantation de PEM avec mémoire partagée. Les interruptions matérielles pour le réseau de stations de travail ont été implantées à l'aide de la librairie C appelée "signal". C'est une approche assez simple et très portable puisque cette librairie est disponible avec beaucoup de systèmes d'exploitation de différents modèles de station de travail. Seulement quelques petits ajustements ont été requis pour porter ParSubC sur des stations de travail DEC, Sun et Hewlett Packard. Malheureusement, cette approche n'est cependant pas disponible avec le Cray T3D qui a plutôt sa propre implantation des interruptions. Ainsi, utiliser la technique d'interruption avec le Cray T3D est une tâche plus complexe qu'utiliser le "polling", et encore plus important, le "polling" a l'avantage d'être une technique beaucoup plus portable. C'est la portabilité du "polling" qui nous a fait préférer cette technique aux interruptions matérielles.

PEM à saveur PMP

Malgré que le protocole choisi pour le partitionnement dans notre implantation soit le PEM, certaines opérations liées au vol de tâche ont été implantées de façon à tirer profit de la mémoire partagée. Notre technique de partitionnement a en quelque sorte une saveur de PMP.

Entre autre, avant d'envoyer une requête de vol de tâche, un processeur vérifie si la pile du processeur ciblé n'est pas vide. Si c'est le cas, il est préférable de passer à un autre processeur, sinon le processeur ciblé sera dérangé inutilement (à moins que pendant l'envoi de la requête le processeur ciblé ait reçu une tâche, ce qui est peu probable...).

```
/* Creation de la tache volee, si disponible */
if (_rtp != _rtt)
{
  _processor *_p = _processor_table[_self()];
  int i;
  _task *tid;
  _word *fp;
  int ra, pc, n1, n2;

  tid = *(_task **)(++_rtp);      /* Identificateur de tache */
  fp = tid->fp;                   /* Pointeur sur les arguments */
  ra = *(int*)(fp-1);             /* Adresse de retour */
  pc = _control_points[ra][0]; /* Point d'entree */
  n1 = _control_points[ra][1]; /* Nombre d'arguments */
  n2 = _control_points[ra][2]; /* Taille du resultat */

  /* Il faut conserver les arguments de la derniere tache volee,
     s'il y en a une, car le voleur n'a peut-etre pas termine de
     les copier. */
  if ((_p->tp-1) != p->ltq)
    _p->fp += _control_points[*(int*)((_p->tp-1)->p->fp-1)][1];

  /* On installe la tache sur la pile d'execution */
  *(_task **)(_fp++) = tid;
  _p->fp += n2;
  *(int *)(_p->fp++) = 0;

  /* Copie des arguments directement de la pile de la victime */
  for (i=0; i<n1; i++) _fp[i] = *fp++;

  return pc; /* On retourne le point d'entree */
}
```

FIG. 5.5 - *Création de tâche par le voleur*

Cette façon de procéder permet d'éviter beaucoup de dérangements inutiles, et dans les cas où il n'y a pas de travail pour tous les processeurs, ce qui arrive lorsque le degré de parallélisme d'un programme est inférieur au nombre de processeurs, cela pourrait faire une différence remarquable.

Une autre distinction à faire avec le PEM pur est que dans notre implantation, c'est le voleur qui crée la tâche et non la victime. Ainsi, la victime n'a simplement qu'à donner le `tid` au voleur, et est donc interrompue moins longtemps. Comme la mémoire est partagée, le voleur peut accéder à toutes les informations dont il a besoin pour la création de la tâche. Dans le PEM pur, le voleur n'a rien à faire pendant que la victime crée la tâche, alors il est grandement préférable de lui faire créer la tâche, ce qui libère

```
{
  _processor *_p = _processor_table[_self()];

  p2 = _p->fp-1;               /* Pointe sur l'adresse de retour */
  _p->fp = (_p->hp-1)->fp; /* On enleve la tache sur la pile */
  p1 = _p->fp+1                /* Pointe sur le debut du resultat */
  fp = tid->fp-1;              /* Pointe sur la fin de l'espace pour
                                  le resultat */

  /* Copie du resultat dans la pile d'execution de la victime */
  while (p2>p1) *--fp = *--p2;
  tid->state = TERMINATED; /* On indique a la victime que la
                              tache volee est terminee */
}
```

FIG. 5.6 - *Retour d'un vol*

la victime plus rapidement. Par souci d'optimisation de l'espace, dans l'implantation pour mémoire distribuée, lorsqu'une victime vole une tâche à son tour, elle ne conserve pas les arguments de la dernière fonction de sa pile de tâches, car ils ne sont plus nécessaires puisqu'elle les a transmis au voleur. Dans la présente implantation, cela n'est pas acceptable puisqu'on ne sait pas quand le voleur a terminé de les copier. Ainsi la victime ne peut plus se permettre d'écrire par-dessus lorsqu'à son tour elle vole une tâche. La création de tâche par le voleur est présentée dans la figure 5.5.

Le retour d'un vol de tâche est un autre exemple de l'efficacité provenant de la mémoire partagée, car le voleur peut effectuer lui-même ce retour, sans assistance de la victime. Comme on peut voir dans la figure 5.6, le voleur n'a simplement qu'à copier le résultat de la fonction, s'il y en a un, directement dans la pile d'exécution de la victime et ensuite signaler la fin d'exécution de la tâche en modifiant l'état de celle-ci. Dans cette figure, le pointeur `p1` pointe sur le début du résultat et `p2` pointe juste au-dessus du résultat. Ainsi le protocole utilisé dans notre implantation est majoritairement à envoi de messages, mais s'inspire aussi de techniques à mémoire partagée.

5.3 CPT basée sur le PEM pour mémoire partagée

Comme l'implantation discutée dans ce chapitre est destinée à une architecture à mémoire partagée, le seul rôle que joue notre PEM est de permettre le partitionnement CPT. On peut donc diviser l'implantation de la CPT basée sur le PEM pour mémoire partagée en trois étapes principales, soit l'envoi d'une requête de vol, la détection de la

```
/* On verifie s'il n'y a pas deja une requete en attente
   de traitement */
if (_p->pending_steal_id == -1)
  {
    _processor *_p = _processor_table[_self()];
    _processor *victim;
    int v = _p->victim_id;

    /* On passe a une autre victime */
    do
      v = (v + 1) %_nb_nodes();
    while (v == _self());
    _p->victim_id = v;
    victim = _processor_table[v];

    lock_processor(v); /* Debut de la section critique */
    /* On verifie que la victime n'a pas deja recu une requete
       de vol et si elle a une tache disponible. */
    if ( (victim->thief_id == -1) && (victim->tp > victim->hp) )
    {
      /* On lui transmet la requete de vol */
      _p->pending_steal_id = v;
      victim->thief_id = _self();
    }
    unlock_processor(v); /* Fin de la section critique */
  }
```

FIG. 5.7 - *Envoi d'une requête de vol*

réception d'une requête et le traitement de la requête de vol.

5.3.1 Envoi d'une requête de vol

Tel que vu dans le chapitre précédent, pour permettre de savoir de qui vient une requête, chacun des n processeurs est identifié par un entier distinct entre 0 et n-*1*. Ainsi lorsqu'un processeur veut envoyer une requête de vol, il suffit de communiquer son numéro d'identification à la victime, et celle-ci sera en mesure de reconnaître quel processeur veut lui voler une tâche.

Cependant, avant d'envoyer sa requête, il est préférable qu'un processeur choisisse bien sa victime. Il doit premièrement s'assurer que celle-ci n'a pas déjà reçu une requête de vol d'un autre processeur, car plutôt que d'attendre que la victime ait traité cette première requête, il est préférable de passer à un autre processeur. De plus, tel que

mentionné précédemment, un processeur choisit toujours une victime qui a au moins une tâche dans sa pile, cela dans le but de diminuer, voir même éliminer les dérangements inutiles.

La figure 5.7 montre le code C associé à l'envoi d'une requête de vol. Trois champs de la structure `processor` sont dédiés au vol de tâche:

- `victim_id`: Ce champ contient le numéro de la prochaine victime potentielle à examiner.
- `pending_steal_id`: Ce champ indique si une requête de vol est en attente de traitement. Si c'est le cas, le champ contient la valeur de la victime qui a reçu la requête en attente, sinon le champ contient -1.
- `thief_id`: Ce champ indique à un processeur qu'il a reçu une requête de vol. Si c'est le cas, le champ contient l'identificateur du voleur, sinon le champ contient -1.

Comme on peut le voir dans la figure 5.7, un processeur peut facilement et rapidement vérifier si un processeur est une bonne victime en regardant simplement la valeur du champ `thief_id` du processeur. Si cette valeur est différente de `-1`, c'est qu'une requête de vol est déjà en attente de traitement par ce processeur, il est donc préférable de passer à un autre processeur. Pour ce qui de s'assurer que le processeur ciblé possède au moins une tâche dans sa pile, il suffit simplement de faire la comparaison `(victim->tp >victim->hp)`. Si cette expression est vraie, c'est qu'il y a au moins une tâche dans la pile de tâches du processeur.

Telle que présentée au chapitre 1, l'utilisation de variables partagées par plusieurs processeurs peut créer des problèmes puisque le résultat est imprévisible. L'envoi d'une requête est un cas qui peut créer des problèmes si elle n'est pas mise dans une section critique. Lorsqu'un voleur a ciblé sa victime et qu'il veut lui envoyer sa requête de vol, il doit s'assurer qu'aucun autre processeur n'envoie une requête de vol à ce même processeur simultanément sinon une des deux requêtes sera perdue. Pour ce, les primitives `lock_processor()` et `unlock_processor()` ont été ajoutées autour du code associé à l'envoi d'une requête (voir la figure 5.7). Grâce à ces primitives, un processeur qui veut exécuter cette section devra attendre que tout autre processeur déjà en train de l'exécuter finisse d'exécuter la section, avant de pouvoir y entrer. De cette façon, il n'y aura qu'un seul processeur à la fois qui pourra exécuter ce code.

5.3.2 Détection de la réception d'une requête de vol

Un élément important pour la performance de la technique de "polling" est l'intervalle entre chaque vérification. Dans notre implantation, il n'y a pas un intervalle de temps fixe à la fin duquel il y a vérification, car cela requierrait une horloge et un mécanisme d'interruption matérielle pour permettre la vérification à intervalles réguliers. Comme le "polling" a été choisi dans le but de ne pas avoir à utiliser les interruptions matérielles, cette solution n'est pas très intéressante. Dans notre implantation, la solution retenue est plutôt d'ajouter des points de "polling" à différents endroits dans le code généré par le compilateur. L'intervalle de "polling" sera donc déterminé par la fréquence de ces points dans le code.

Il n'est pas possible d'avoir une fréquence idéale, car cela est différent d'un programme à l'autre. Cependant, il est important que le nombre de points soit suffisant pour que le délai de traitement d'une requête de vol ne soit pas trop élevé. Il est aussi très important de garder aussi bas que possible le coût associé aux points de "polling", de cette façon si la fréquence des points de "polling" est plus élevé que nécessaire, les performances n'en seront pas trop affectées.

Une première question est qu'est-ce qu'un point de "polling" dans notre implantation? Lorsqu'un voleur veut faire un vol de tâche, il le signale à sa victime en faisant l'affection du champ `thief` de la victime avec son identificateur:

```
victim->thief_id = _self();
```

Comme initialement et après chaque vol de tâche, une victime met ce champ à la valeur -1, une victime peut donc vérifier si elle a reçu une requête de vol par le simple test suivant:

```
if (_p->thief_id>=0) _poll();
```

Le "polling" peut donc se faire de façon très efficace, car avertir un processeur de son intention de vol se fait en une simple affectation et l'opération de "polling" se limite à faire une simple comparaison d'entier et un branchement. Le dernier problème qu'il reste désormais à résoudre pour avoir une implantation efficace de "polling" est de déterminer où mettre les points de "polling" dans le code.

Emplacement des points de "polling"

Dans notre implantation, l'intervalle de "polling" sera déterminé par la fréquence des points de "polling" dans le code généré. Malgré qu'il n'existe pas un intervalle

parfait commun à tous les programmes, il faut pour avoir une implantation efficace que ces points soient assez nombreux pour que le temps de réponse d'une requête de vol de tâche soit petit, sans toutefois que les processeurs effectuent trop de vérifications inutiles.

Par exemple, ne mettre des points de "polling" qu'à l'entrée des fonctions ne seraient guère suffisant puisqu'un programme peut contenir des boucles qui peuvent consommer beaucoup de temps sans pour autant que ces boucles ne contiennent d'appel de fonction. Ainsi le temps de réponse pour une requête de vol de tâche lorsque la victime exécute une telle boucle serait élevé et cela pourrait diminuer de beaucoup les performances.

Complètement à l'opposé, on retrouve la solution qui consisterait à mettre un point de "polling" à chaque instruction. Cette approche offre un temps de réponse minimal, par contre, lorsque les processeurs exécutent une tâche, ils passent la moitié de leur temps à vérifier s'ils ont reçu une requête de vol. Le temps passé à la vérification des requêtes est donc beaucoup trop élevé pour avoir une implantation efficace. Il faut donc trouver une solution efficace qui se situe entre ces deux extrêmes.

Ce qu'il faut principalement s'assurer est de mettre des points de "polling" aux endroits qui peuvent avoir un temps d'exécution élevé. Par exemple, une boucle `for` est une structure qui peut consommer beaucoup de temps de calcul, il est donc préférable de mettre un point de "polling" dans la boucle de telle façon qu'à chaque tour de boucle, il y ait vérification pour des requêtes de vol en attente.

Il faut donc identifier toutes les parties d'un programme qui peuvent constituer une forme de boucle pour y mettre un point de "polling" à l'intérieur. En C, on retrouve dans cette catégorie, les boucles `for`, `while` et `do-while`. Il faut également inclure dans cette catégorie les étiquettes, car elles peuvent facilement être utilisées pour construire des boucles, et peuvent donc aussi résulter en un temps de calcul élevé.

La dernière structure en C qui peut consommer beaucoup de temps de calcul est l'appel de fonction. Lorsqu'utilisé de façon récursive, l'appel de fonction peut en quelque sorte constituer une forme de boucle et par le fait même consommer beaucoup de temps de calcul. Un bel exemple de programme qui utilise ce style de boucle est le programme `pfib.c`, présenté dans la figure 5.4 de la page 77, qui calcule la suite de Fibonacci. Comme les appels de fonctions récursifs peuvent consommer à eux seuls beaucoup de temps de calcul, il est préférable de mettre un point de "polling" à l'entrée des fonctions. On peut donc résumer les formes qui sont ou peuvent constituer une boucle et qui

```
void _poll()
{
  _processor *_p = _processor_table[_self()];
  int t = _p->thief_id;
  _processor *thief = _processor_table[t];

  /* On verifie s'il y a une tache disponible */
  if (_p->hp < _p->tp)
  {
    _task *tid = _p->hp;

    /* On copie la tache dans le RTQ du voleur */
    tid->state = STOLEN;
    _p->hp++;
    *(_task **)thief->rtp = tid;
    thief->rtp--;
  }

   /* On indique la fin du traitement de la requete */
  _p->thief_id = -1;
  thief->pending_steal_id = -1;
}
```

FIG. 5.8 - *"polling"*

peuvent donc prendre un temps de calcul élevé à:

- Les boucles `while`, `do-while` et `for`.
- Les étiquettes
- Les fonctions

Pour éviter des intervalles de "polling" trops grands, ces formes contiendront donc toutes un point de "polling".

5.3.3 Traitement d'une requête

Quand un processeur s'aperçoit à un point de "polling" qu'il a reçu une requête de vol, il exécute alors immédiatement le code associé au traitement des requêtes (voir la figure 5.8). S'il possède une tâche ou plus dans sa pile, il envoie alors le `tid` de la tâche du dessous dans la pile de tâche prête du voleur (RTQ), et indique que cette tâche a été volée. Il enlève également cette tâche de la pile (ou dans ce cas-ci de la queue). La

victime doit ensuite indiquer qu'elle est prête à recevoir d'autres requêtes, et avertir le voleur que sa requête a été traitée.

Il est à noter que les processeurs possèdent une pile de tâches prêtes à exécuter (RTQ) uniquement par analogie avec l'implantation avec mémoire distribuée, car il n'y aura jamais plus d'une tâche dans cette pile.

5.4 Résumé

Ce qui distingue principalement l'implantation avec mémoire partagée par rapport à celle avec mémoire distribuée, est le moyen utilisé par les processeurs pour communiquer. Tel que vu dans ce chapitre, grâce à la mémoire partagée, les processeurs possèdent un moyen de communication très simple et très efficace, soit celui d'écrire et de lire en mémoire. Le seul mécanisme à implanter fut la CPT qui a été basée sur un protocole à envoi de messages afin de garder le plus de similitudes possibles entre les deux implantations.

Malgré que la CPT soit basée sur un PEM, les processeurs tirent avantage de la mémoire partagée pour mieux choisir leur victime lors d'un vol de tâche. L'accès aux mêmes données permet aussi aux processeurs d'effectuer le retour de vol sans l'intervention de la victime. Simplicité et efficacité sont deux qualificatifs typiques d'une architecture à mémoire partagée.

Chapitre 6

Évaluation

6.1 Mesures et méthodes d'évaluation

Les programmes utilisés pour évaluer le langage ParSubC et son implantation pour le Cray T3D et le réseau de stations de travail HP, sont des traductions en ParSubC de programmes initialement écrits en Multilisp. Ces programmes ont été utilisés pour évaluer une implantation de Multilisp basée sur la création paresseuse de tâches [Fee93]. Ces programmes sont tous de petites tailles (< 200 lignes de code ParSubC) et sont tous basés sur des algorithmes *diviser-pour-régner* parallèles. Certains programmes, basés sur la programmation fonctionnelle, sont résolus strictement par appels de fonction, alors que d'autres basés sur la programmation impérative ont besoin d'une mémoire partagée. L'annexe A décrit brièvement chacun des programmes, en plus de présenter le code lui-même.

Chacun des programmes a été exécuté plusieurs fois sur n=1 à 16 processeurs et la moyenne des temps d'exécution pour chaque n a été calculée. Pour les deux implantations, 16 était le nombre maximum de processeurs avec lesquels les tests pouvaient être exécutés. Pour le Cray T3D, cela vient d'une contrainte matérielle provenant de l'architecture, qui limite à 16 le nombre de processeurs qui peuvent être configurés pour avoir une mémoire partagée. Pour le réseau de stations de travail HP, il aurait été possible d'avoir plus de 16 stations. Cependant, pour avoir des résultats significatifs, il est préférable d'utiliser des stations de même performance. Cela a limité le nombre de stations disponibles pour l'exécution parallèle des programmes à 16.

Les temps d'exécution des programmes sur un nombre variable de processeurs a

permis de calculer l'accélération qu'il est possible d'obtenir avec ParSubC. L'accélération calculée est relative car nous voulions, dans un premier temps, évaluer uniquement les performances possibles avec les programmes parallèles ParSubC, sans se soucier du coût associé à la parallélisation de ceux-ci. Ce coût sera évalué séparément. L'accélération avec n processeurs sera donc calculée avec le temps d'exécution sur un processeur, T_1, et le temps d'exécution sur n processeurs, T_n :

$$A_n = \frac{T_1}{T_n}$$

Pour obtenir une appréciation honnête de ParSubC et de ses implantations, il ne faut par uniquement regarder les accélérations relatives obtenues, mais il faut aussi évaluer le coût associé à l'utilisation du parallélisme. Pour évaluer ce coût, deux autres mesures seront calculées. Premièrement, le temps pris pour l'exécution des programmes sans l'opérateur `!!`, $T_1^{Sans!!}$, sur 1 processeur sera calculé. Nous pourrons donc voir le coût associé à la gestion des tâches (empilement et dépilement). Ensuite, des versions séquentielles de certains programmes ont été écrits, et leurs temps d'exécution ont été mesurés. Les versions séquentielles utilisent des boucles aux endroits où les versions parallèles utilisent des algorithmes diviser-pour-régner. Ces programmes sont également présentés dans l'annexe A. Pour bien évaluer le coût de la parallélisation d'un programme avec ParSubC, les temps pris pour exécuter ces versions séquentielles, T_{seq}, seront comparés à ceux des algorithmes parallèles sans `!!` sur un processeur. Le coût associé à l'utilisation du parallélisme sera donc évalué à l'aide des deux facteurs suivants:

- Gestion de tâches: $G = \frac{T_1}{T_1^{Sans!!}}$
- Parallélisation: $P = \frac{T_1^{Sans!!}}{T_{seq}}$

Une autre mesure qu'il est intéressant de prendre est le coût associé au dérangement par les processeurs libres lorsque le degré de parallélisme est inférieur au nombre de processeurs disponibles. Dans ce cas, il y a un ou plusieurs processeurs qui passent leur temps à essayer de voler une tâche et cela peut ralentir les processeurs qui travaillent. Pour calculer ce facteur de dérangement, les versions sans `!!` des programmes ont été exécutées sur 1 à 16 processeurs. Avec ces temps, le facteur de dérangement peut se calculer comme suit:

$$F = \frac{T_{16}^{Sans!!}}{T_1^{Sans!!}}$$

Nous allons également mesurer le surcoût des instructions associées aux données partagées que l'on retrouve avec l'implantation pour mémoire distribuée. Plus précisément, le surcoût pour l'écriture à une variable globale et celui des vérifications associées aux qualificatifs `light` et `heavy`. Ceci sera évalué en comparant les temps d'exécution

des programmes parallèles sur 1 processeur à ceux de variantes sans instruction pour donnnées partagées sur 1 processeur.

Un dernier point de notre implantation à évaluer est la qualité du compilateur ParSubC en tant que compilateur C. Pour cela, une version des programmes sans !! et sans instruction pour données partagées sera compilée avec un compilateur natif C et les temps d'exécution obtenus seront comparés à ceux obtenus avec les programmes compilés avec le compilateur ParSubC. Cette comparaison des temps d'exécution nous permettra d'évaluer le coût associé à la génération du code C intermédiaire de notre compilateur plutôt que du code machine. Cette mesure n'est prise qu'à titre informatif, puisque notre compilateur n'est qu'un prototype et que peu d'efforts ont été mis à l'optimisation du code C généré.

6.1.1 Loi d'Amdahl

Un phénomène intéressant à prendre en considération lorsqu'on évalue l'accélération de programmes parallèles est la loi d'Amdahl [Amd67] [Wil95]. Selon la loi d'Amdahl, tout programme possède une accélération maximale. Cette loi présente un programme parallèle comme possédant une portion séquentielle et une portion parallèle. Si P_{Seq} désigne la portion séquentielle d'un programme, alors par conséquent $(1 - P_{Seq})$ désigne la portion parallèle. Ainsi, si l'on reprend la formule de l'accélération, on peut exprimer l'accélération maximale d'un programme avec n processeurs comme suit:

$$A_n = \frac{T_1}{T_n} = \frac{T_1}{P_{Seq}T_1 + \frac{(1-P_{Seq})T_1}{n}} = \frac{1}{P_{Seq} + \frac{1-P_{Seq}}{n}}$$

C'est donc dire, qu'à moins que la portion séquentielle d'un programme ne soit nulle, l'accélération ne sera pas linéaire, mais tendra plutôt, au fur et à mesure que le nombre de processeurs augmente, vers:

$$A_{max} = \frac{1}{P_{Seq}}$$

6.2 Performances du Cray T3D

De façon générale, les accélérations obtenues avec l'exécution des 7 programmes ParSubC sont assez bonnes. Les temps d'exécution sur 1 processeur ainsi que les accélérations obtenues sont présentés dans la table 6.1. On retrouve également dans cette table la fréquence d'accès aux variables partagées. Cette donnée est utile pour expliquer certains résultats. Elle se définit comme suit:

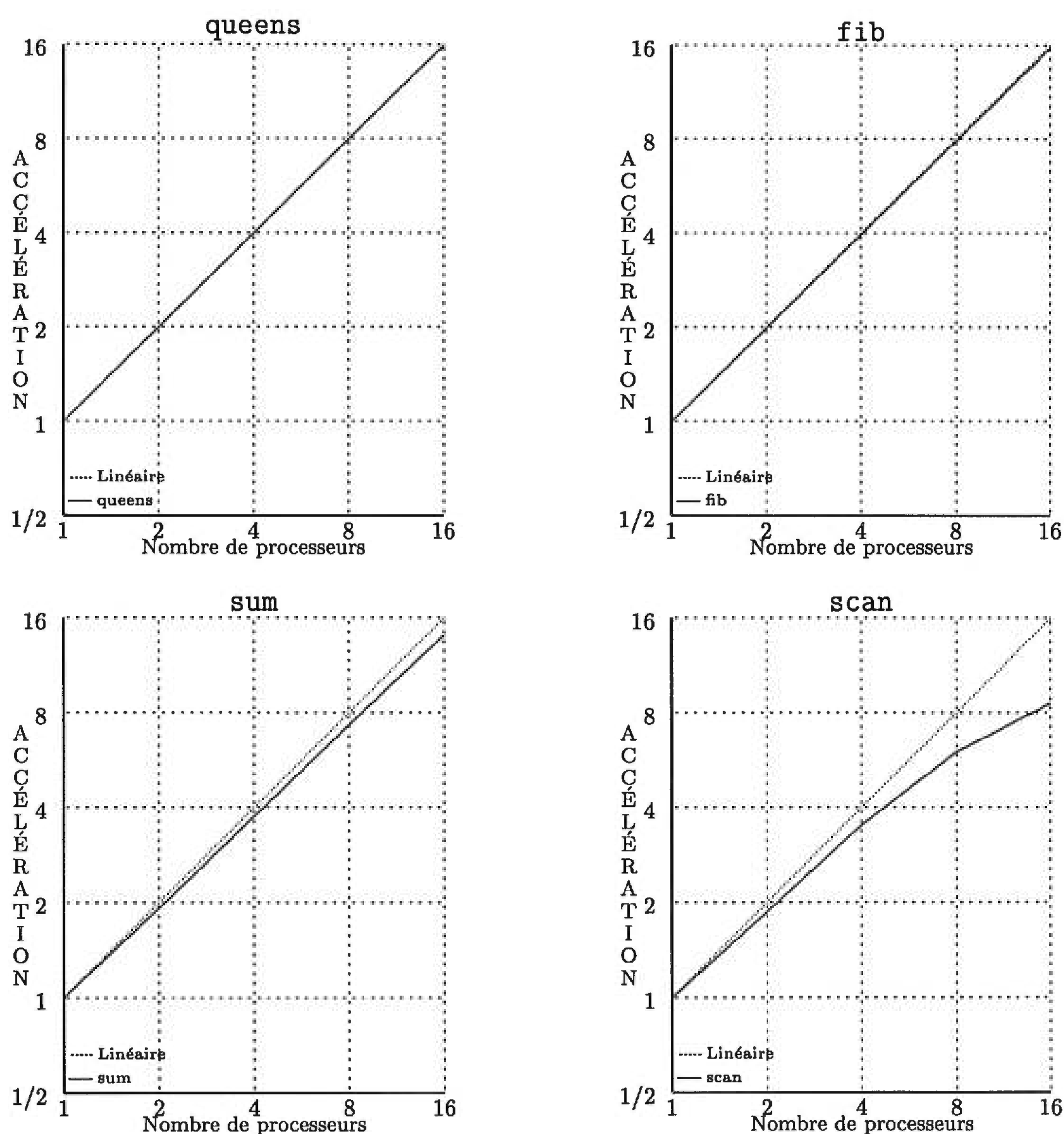

FIG. 6.1 - *Courbes d'accélération pour* queens, fib, sum *et* scan — *Cray T3D*

Programmes	T_1 (sec.)	f ($\times 10^5$)	Accélérations			
			A_2	A_4	A_8	A_{16}
`queens`	10.22	0	2.00	4.00	7.99	15.94
`fib`	5.66	0	1.99	3.96	7.90	15.72
`sum`	3.00	1.67	1.92	3.75	7.33	14.13
`scan`	1.72	3.81	1.87	3.52	6.02	8.58
`poly`	8.06	9.93	1.70	3.00	5.21	6.08
`mm`	9.03	7.50	1.70	3.07	4.71	4.80
`abisort`	4.62	10.2	1.65	2.62	3.39	3.63

TAB. 6.1 - *Accélérations avec le Cray T3D.*

$$f = \frac{\text{Nombre d'accès aux variables globales}}{T_1}$$

Une représentation graphique des accélérations est également disponible sous forme de courbes d'accélération dans les figures 6.1 et 6.2. D'après les résultats de la table 6.1, on peut classer les programmes exécutés dans trois catégories:

- Excellents: Les programmes ayant donné d'excellents résultats sont `queens`, `fib` et `sum`. Dans les deux premiers cas, on peut même presque parler d'accélérations linéaires. Ces deux programmes sont résolus uniquement par appels de fonction et n'utilisent aucunement la mémoire partagée. Le troisième programme donne un résultat un peu moins bon et ce, probablement à cause des accès à la mémoire partagée (chaque donnée du vecteur doit être lue). Il faut se rappeler que la mémoire partagée est de type NUMA, et que toutes les variables globales sont dans la mémoire du processeur 0. Sauf pour le processeur 0, la lecture des éléments du vecteur n'est donc pas une lecture en mémoire locale, et requiert un accès au réseau d'interconnexion (qui est 3 à 4 fois plus lent qu'un accès local).

- Moyens: Deux des 7 programmes exécutés ont donné des résultats moyens, soit `scan` et `poly`. La raison qui explique que ces deux programmes aient donné de moins bons résultats est la fréquence plus élevée des accès aux variables globales (et donc une fréquence plus élevée de communications inter-processeur). Dans le cas de `sum`, la fréquence n'est pas nulle comme c'est le cas avec `queens` et `fib`, mais elle est quand même assez basse pour permettre d'avoir de bonnes accélérations. Pour ce qui est de `scan` et `poly`, la fréquence devient un facteur qui nuit considérablement aux performances.

- Pauvres: Les deux derniers programmes, soit `mm` et `abisort` ont donné des résultats

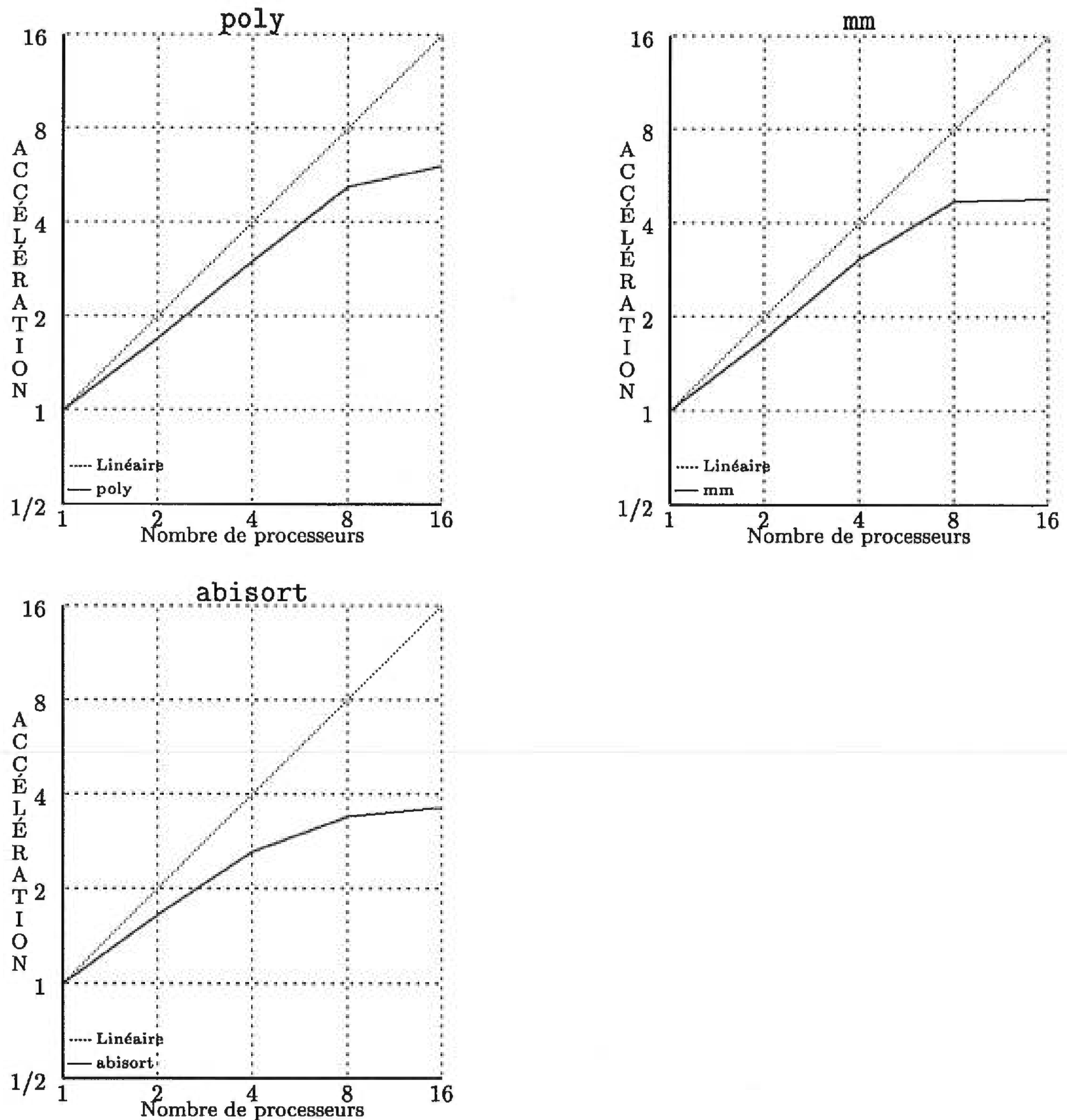

FIG. 6.2 - *Courbes d'accélération pour* poly, mm *et* abisort — *Cray T3D*

plutôt pauvres. L'explication provient encore de la fréquence d'accès aux variables globales. Celle-ci étant encore plus élevée pour ces deux derniers programmes, les performances en sont d'autant plus affectées.

Les programmes ParSubC exécutés sur le Cray T3D ont donc tous offert une accélération, mais les accès non-locaux étant 3 à 4 fois plus lents que les accès locaux, les programmes qui exigeaient beaucoup de références aux variables globales ont des

Programmes	T_1	Accélérations			
		A_2	A_4	A_8	A_{16}
`queens`	2.05	1.93	3.52	5.25	5.76
`poly`	1.75	1.93	3.38	4.95	5.42
`sum`	0.64	1.96	3.10	4.44	4.92
`fib`	1.33	1.81	3.06	4.23	3.74
`mm`	2.25	0.37			
`abisort`	3.37	0.13	$\ll 1$		
`scan`	1.22	0.04			

TAB. 6.2 - *Accélérations avec le réseau de stations HP.*

accélérations moins élevées.

6.3 Performances avec le réseau de stations HP

Lorsqu'on compare les accélérations de la table 6.2 par rapport à la table 6.1, on constate que les accélérations obtenues avec l'implantation de ParSubC pour le réseau de stations de travail sont beaucoup moins bonnes. Les accélérations obtenues sont toutes plus petites que 6, et pire encore, trois programmes n'ont pas d'accélération lorsqu'on augmente le nombre de stations.

La principale cause d'inefficacité est la lenteur de la mémoire partagée virtuelle. Ainsi tous les programmes reposant fortement sur une mémoire partagée ont vu leur performance diminuer lors de l'augmentation du nombre de stations. C'est ce qui est arrivé pour les programmes `mm`, `abisort` et `scan`. Lorsqu'on passe de 1 à 2 stations, le temps d'exécution est 3 fois plus élevé dans le cas de `mm`, 8 fois plus élevé pour `abisort` et 28 fois plus élevé pour `scan`. Cette décélération s'accentue au fur et à mesure qu'augmente le nombre de stations, ce pourquoi les accélérations avec 4, 8 et 16 stations ne sont pas présentées dans la table 6.1 (pour certains cas, les temps d'exécution se comptent en minutes). Avec les profils d'exécution de l'annexe B, on peut voir le temps pris par les accès à la MPV.

Une distinction à faire cependant au sujet de la lenteur de la MPV est par rapport au type d'accès. Si les programmes `mm`, `abisort` et `scan` ont de si mauvais résultats avec la MPV, c'est que ce sont tous des programmes qui ont une fréquence élevée d'affectations à des variables globales, ce qui cause des diffusions et nécessite des accès au réseau. Ainsi

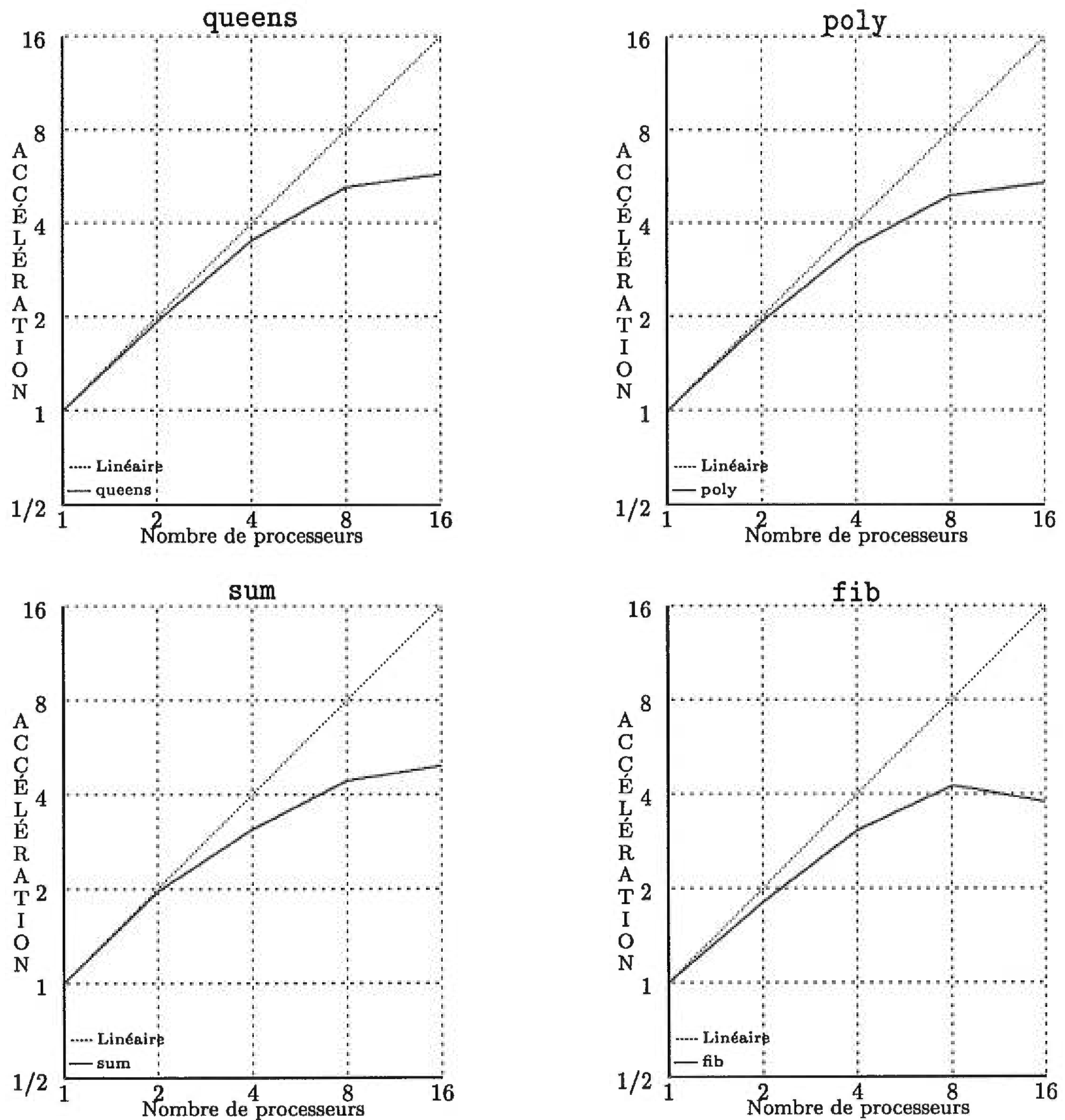

FIG. 6.3 - *Courbes d'accélération de* queens, poly, sum *et* fib — *Multi-ordinateur*

les programmes poly et sum offrent une accélération même s'ils utilisent fortement un vecteur global, car ils ne font que des lectures. Dans ces cas, il n'y pas d'accès au réseau puisqu'une lecture à une variable globale ne consiste qu'en une lecture locale.

Si la MPV explique les mauvaises performances des programmes mm, abisort et scan, cette explication n'est plus valable dans le cas des 4 autres programmes, puisqu'ils ne font aucune affectation à la mémoire partagée. Pour ces programmes, c'est plutôt la

granularité des tâches qui explique que les accélérations soient toutes inférieures à 6. Lorsque l'on regarde les courbes d'accélérations présentées dans la figure 6.3, on peut voir que les courbes présentent quand même une accélération. On peut voir l'effet de la loi d'Amdahl qui diminue la pente de la courbe lorsque le nombre de stations augmente, mais au moins il y a accélération. Comme les communications inter-processeurs sont faites à l'aide d'un réseau Ethernet pour les stations de travail, le partitionnement est plus lent que pour le Cray T3D, et par conséquent, la granularité des tâches doit être plus grosse pour arriver à des performances égales à celles du Cray T3D. Si la granularité des tâches était augmentée, cela retarderait l'aplatissement de la courbe, et ainsi la courbe d'accélération jusqu'à 16 stations tendrait plus vers l'accélération linéaire.

À titre d'exemple, nous avons exécuté le programme `fib` avec $n = 35$ plutôt que $n = 30$, et le programme queens avec $n = 13$ plutôt que $n = 12$. Les temps d'exécution sur 1 station sont respectivement passés de 1.33 à 14.64 secondes et de 2.05 à 11.25 secondes. Ce qui est particulièrement intéressant c'est qu'avec 16 stations, les accélérations sont alors passées de 3.74 à 11.56 et de 5.76 à 11.68, ce qui est beaucoup mieux. Les nouvelles courbes d'accélération pour ces problèmes sont illustrées dans la figure 6.5 à la page 105. En augmentant la granularité des tâches, on peut donc obtenir avec le multi-ordinateur des résultats semblables à ceux du Cray T3D sur certains programmes.

6.3.1 Programmes avec migration de tâches

Jusqu'à présent, les programmes présentés pour évaluer le potentiel de ParSubC sont basés uniquement sur la migration de données. Comme certains de ces programmes ont une fréquence élevée d'écritures sur des données globales (et donc d'accès au réseau), il convient de se demander s'il ne serait pas plus avantageux d'utiliser la migration de tâches. Pour cela, au lieu d'utiliser des structures globales, les données seront réparties à travers les stations. Ainsi, au lieu de provoquer plusieurs migrations de données, la tâche sera plutôt migrée vers la station où se trouve ces données.

Ce ne sont cependant pas tous les programmes qui sont de bons candidats à la migration de tâches, car dépendant de l'algorithme, les tâches pourraient être migrées très souvent et, dans certains cas, trop souvent pour offrir de bons résultats. Un exemple de programme pour qui la migration de tâches ne peut améliorer les résultats est `mm`. Le problème est que l'algorithme ne travaille pas localement sur les matrices: pour chaque élement de la matrice résultante à calculer, il faut lire une colonne et une rangée au complet des matrices à multiplier. Il n'existe pas de partitionnement qui puisse faire travailler l'algorithme sur les données locales à la station seulement, et donc la tâche

sera migrée plusieurs fois pour chaque élément à calculer. Un programme qui consisterait plutôt à faire l'addition de deux matrices serait un très bon candidat pour la migration de tâches, puisque le calcul de l'élément xy ne requiert que la lecture des éléments xy des deux matrices à additionner.

On peut énoncer les caractéristiques requises pour qu'un programme soit avantagé par la migration de tâches comme suit:

- Fréquence élevée d'écritures sur des données partagées.
- Algorithme travaillant localement sur les données partagées. Il est à noter qu'un bon partitionnement de données peut résoudre les problèmes de localité d'un algorithme.

Parmi les programmes présentés jusqu'à présent, seul `scan` est un bon candidat pour la migration de tâches. Cependant, pour avoir plus de données pour nous permettre d'évaluer la migration de données à la migration de tâches, nous avons également écrit une variante de `sum`, appelée `sum_mt`, à l'aide de `heavy`. Ce qui fait de `sum` un programme moins intéressant pour la migration de tâches est qu'il ne fait que lire le vecteur global (ce qui peut être fait sans communication réseau par toutes les stations). De plus, un nouveau programme a également été écrit. Le programme `square` replace chaque élément d'un vecteur par son carré. Dans chacun des trois programmes, les vecteurs sont distribués de sorte que chacune des n stations exécutant le programme possède un bloc constituant un $n^{\text{ième}}$ du vecteur. Les trois programmes utilisant l'annotation `heavy` ainsi que leurs profils d'exécution sur 16 stations sont présentés dans les annexes de ce document.

Résultats obtenus

L'exécution de la première écriture des programmes avec migration de tâche fut très décevante. Il fut impossible d'obtenir des résultats car ces programmes causent un nombre de migrations de tâches trop élevé pour la taille des structures de la CPT de l'implantation pour stations de travail. Le problème de ces programmes est que les accès aux pointeurs annotés `heavy` sont situés après l'opérateur parallèle `!!`. Cela implique que lorsqu'une station exécute une fonction contenant `!!`, elle ira jusqu'au dernier appel récursif avant d'accéder un pointeur `heavy`. À ce point, si le pointeur référence une donnée qui ne lui appartient pas, le dernier appel de fonction est migré. Le problème

Programmes	T_1	Accélérations			
		A_2	A_4	A_8	A_{16}
`square`	0.58	1.98 1.55	3.79 2.58	7.00 4.52	9.48 7.74
`sum_mt`	0.27	1.59 0.95	3.23 2.79	5.77 3.98	6.16 4.30
`scan_mt`	1.68	1.93 1.36	3.41 1.37	5.09 1.98	3.25 2.55

TAB. 6.3 - *Performances avec la migration de tâches*

est que tous les appels récursifs référant à un élément de la portion du tableau qui ne lui appartient pas devront éventuellement être migrés.

Si on prend par exemple le cas de `sum_mt` avec deux stations, chaque station possède la moitié du vecteur, soit 250 000 éléments. Donc 250 000 tâches devront être migrées car chaque feuille de l'arbre causera une migration. Ce nombre est trop élevé pour la taille des structures de la CPT disponible dans notre implantation. Il n'est pas vraiment utile d'essayer d'augmenter la taille des structures de la CPT car il y aurait de toute façon beaucoup trop de migrations de tâches pour obtenir des accélérations. Il faut donc de toute évidence apporter des modifications aux programmes si on veut obtenir des résultats intéressants. Deux approches ont été utilisées dans les programmes pour éviter ce problème:

- Modifier l'algorithme pour qu'il n'effectue pas de récursivité jusqu'aux éléments, mais qu'il travaille plutôt sur des blocs d'élements. Ceci limitera ainsi le nombre de migrations de tâches à un nombre plus raisonnable. Des modifications ont été apportées aux programmes `sum` et `square` pour utiliser cette approche.
- Ajouter une référence au pointeur annoté `heavy` avant l'opérateur `!!`. C'est ce qui est déjà fait dans la fonction `scan2` du programme `scan_mt`. Nous avons cependant du en ajouter une artificielle dans la fonction `scan2` car c'est cette fonction qui causait le dépassement de la taille des structures. Cette solution n'a cependant été retenue que pour des fins de comparaison, car elle est très dépendante de l'implantation de ParSubC, ce que nous voulons éviter de voir dans le langage ParSubC.

Les accélérations obtenues avec la deuxième version des programmes `scan_mt`, `sum_mt` et `square` sont présentées dans la table 6.3 et les courbes résultantes dans la figure 6.4. Les accélérations sont présentées en paire, le premier chiffre étant la meilleure accélération de 8 exécutions et le deuxième étant l'accélération moyenne. Nous avons effectué 8

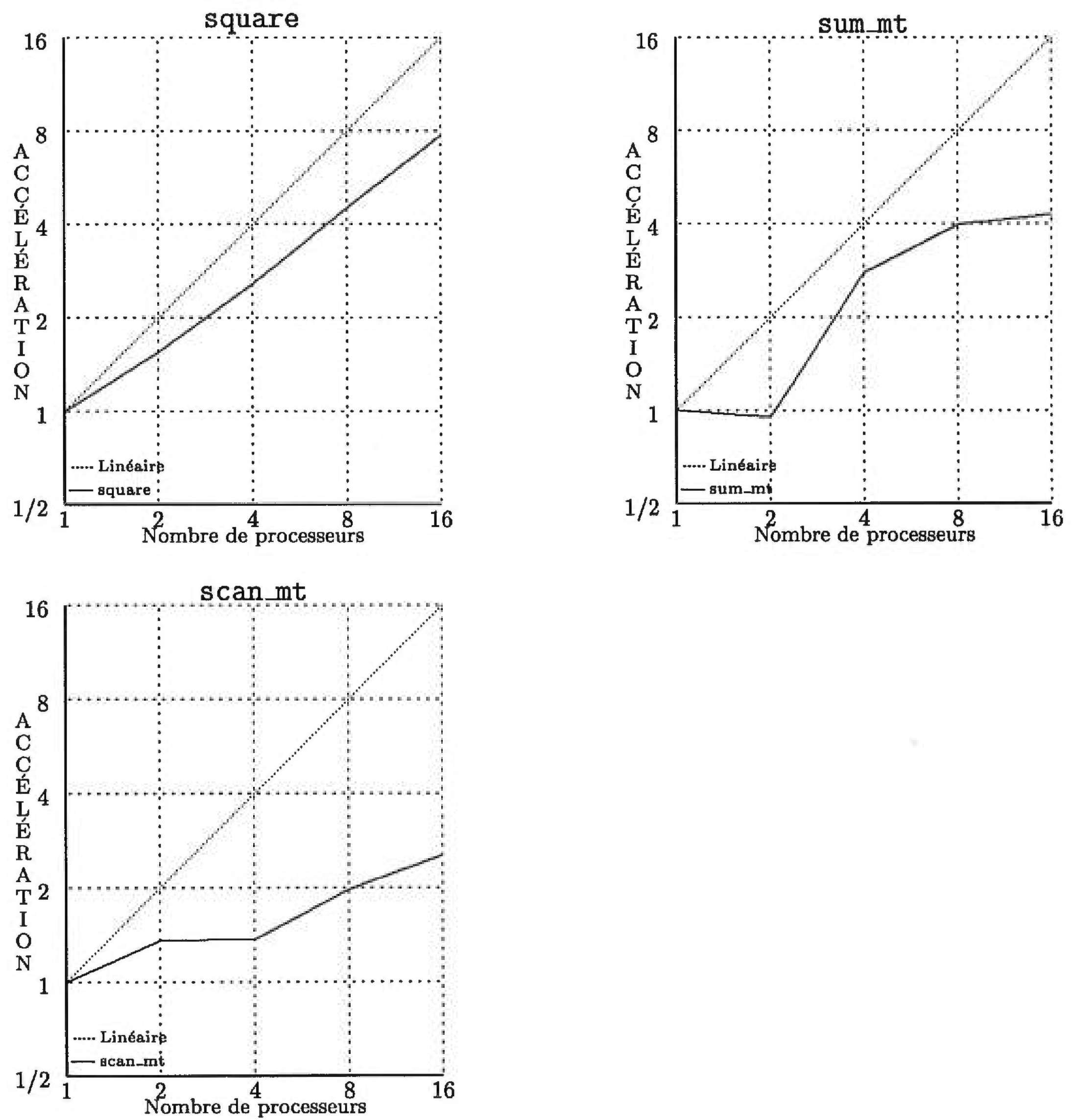

FIG. 6.4 - *Courbes d'accélération pour* sum_mt, scan_mt *et* square

exécutions pour la migration de tâches, plutôt que 3 pour la migration de données, car les résultats étaient beaucoup plus instables qu'avec la migration de données.

Pour commencer, si l'on s'attarde aux meilleures accélérations (premier chiffre), on peut voir que les trois programmes offrent de bonnes accélérations avec 2 et 4 stations. Avec 8 stations, seulement `square` offre une bonne accélération, `sum_mt` offre une accélération moyenne et `scan_mt` une faible accélération. Avec 16 stations, les résultats sont encore moins bons: `square` ne présente qu'une accélération moyenne, `sum_mt` une faible accélération et `scan_mt` une très faible accélération.

Le problème commun de ces trois programmes est qu'ils ont tous une granularité trop fine pour avoir de bonnes accélérations avec un nombre élevé de stations. Le problème se présente plus tôt pour `sum_mt` et `scan_mt`, mais il se retrouve avec tous les programmes avec 16 stations. Malgré cela, les résultats démontrent qu'il est possible d'avoir de bonnes accélérations avec la migration de tâches. Les programmes `square` et `scan_mt` sont d'autant plus intéressants puisque la migration de tâches a permis d'obtenir des accélérations alors que la migration de données n'en offre pas.

Instabilité des performances

Une particularité des résultats obtenus avec la migration de tâches est leur instabilité. Avec des exécutions parallèles sur un réseau de stations, il ne faut pas s'attendre à obtenir des résultats toujours constants, principalement à cause du réseau qui est une ressource partagée. Nous avons exécuté les programmes au moment le moins achalandé, mais ParSubC n'en avait pas l'exclusivité, d'où la présence de petites variances dans les résultats des exécutions parallèles pour les programmes avec migration de données. Cependant, avec la migration de tâches, les écarts entre les résultats sont beaucoup plus importants. Le pire exemple que nous avons rencontré est avec le programme `scan_mt` avec 2 stations: le meilleur temps d'exécution est de 0.87 et le pire est de 3.53, soit un écart d'un peu plus de 400 pour cent. C'est à cause de ces grands écarts que nous avons décidé de présenter, dans la table 6.3, les meilleurs accélérations en plus des accélérations moyennes. Ainsi, on peut donc constater qu'avec la migration de tâches, un autre facteur vient pertuber les résultats de façon beaucoup plus importante.

Avec les trois programmes présentés, le travail des tâches consiste à faire un calcul sur une portion d'un tableau. Comme chacune des portions appartient à une station, seulement cette station peut exécuter une tâche, du moins au complet, ce qui introduit en quelque sorte le concept de *propriétaire de tâche*. On retrouve donc maintenant un facteur de probabilité dans l'exécution des tâches. La meilleur exécution est celle

où toutes les tâches qui ont été volées, l'ont été par leur propriétaire. À l'opposé, on retrouve le cas où chaque tâche est, dans un premier temps, volée par une autre station que son propriétaire pour ensuite devoir être migrée. Dans le meilleur cas, on a donc aucune migration de tâches contrairement à une à chaque tâche volée dans le pire cas. De plus, dans ce dernier cas les stations se retrouvent donc en recherche de travail deux fois plus souvent que dans le meilleur cas.

On peut voir dans l'annexe B le temps passé sans travail par les stations avec un exemple de bonne et de mauvaise exécution de `scan_mt` avec deux stations. Lorsque l'on regarde les résultats de ce dernier exemple, on observe qu'il ne semble y avoir qu'une seule station sur deux qui travaille pendant une très grande partie de l'exécution. Ceci peut peut-être s'expliquer par un phénomène d'*emprisonnement de tâche*. Lors de la rencontre de l'opérateur `!!`, la station empile une tâche qui travaille sur la portion dont elle est le propriétaire et commence l'exécution d'une fonction qui travaille sur l'autre portion du tableau (dont elle n'est pas propriétaire). La station reçoit une requête de vol et retourne donc la tâche empilée. Elle continue l'exécution de sa tâche courante et rencontre une expression qui force la migration de la fonction vers l'autre station. Le voleur reçoit donc une tâche volée suivie d'une tâche migrée. Le problème surgit lorsque le voleur reçoit la tâche migrée avant d'avoir commencé l'exécution de la tâche volée. Dans ce cas, il exécutera la tâche migrée en premier, et ce n'est qu'après l'exécution de celle-ci, qu'il commencera l'exécution de la tâche volée pour réaliser que celle-ci doit être migrée. Dans un tel cas, l'exécution des deux tâches se fait donc de façon séquentielle.

Partitionnement des données

Jusqu'à présent les programmes avec migration de tâches présentés n'utilisaient qu'un seul type de partitionnement de données, qui consiste à diviser le vecteur en n blocs et à en donner un à chacune des n stations. Il serait intéressant de voir les résultats possibles avec d'autres types de partitionnment de données. Pour cela, nous avons écrit une variante de `square` avec laquelle nous varions le nombre de blocs et dont l'allocation des blocs se fait de façon alternée entre les stations. Nous avons exécuté cette variante avec 1, 2, 4, 8 et 16 stations. Il est à noter que toutes les accélérations ont été calculées avec le meilleur temps (0.58 seconde) d'exécution avec 1 station de tous les types de partitionnement de données, et non le temps d'exécution avec 1 station de chacun d'eux.

D'après les accélérations présentées dans le tableau 6.4, on voit que le meilleur partitionnement de données est celui d'un bloc par station. Plus on augmente le nombre

Nombre de blocs	Accélérations							
	A_2		A_4		A_8		A_{16}	
n	1.98	1.55	3.79	2.58	7.00	4.52	9.48	7.74
16	1.83	1.13	3.49	2.83	5.58	3.89	10.18	5.80
32	1.58	0.99	2.87	2.52	5.22	4.39	8.29	6.59
64	0.32	0.26	1.24	1.19	2.76	2.42	4.26	3.54
128	0.41	0.38	1.27	1.12	2.26	2.13	3.87	3.41
256	0.44	0.43	1.37	1.31	2.01	1.81	3.30	3.09

TAB. 6.4 - `square` *avec différents partitionnements de données*

de blocs, moins bonnes sont les accélérations. L'avantage potentiel de l'augmentation du nombre de bloc pourrait être d'augmenter le nombre de tâches pour ainsi accélérer le partitionnement de tâches. Cependant, comme avec la migration de tâches les tâches ont en quelque sorte un propriétaire, l'augmentation du nombre de blocs a surtout pour effet d'augmenter le nombre de migrations de tâches, ce qui diminue les performances.

On peut voir encore une fois l'instabilité des résultats avec les partitionnements en n et 16 blocs. Ces deux partitionnements de données devraient donner les mêmes résultats avec 16 stations, mais ce n'est cependant pas ce qu'on retrouve dans la table 6.4.

6.3.2 Migration de données *vs* migration de tâches

Si on compare les résultats de `sum_mt` obtenus avec la migration de tâches présentés dans la table 6.3 avec ceux de `sum` avec la migration de données présentés dans la table 6.2, on remarque que la migration de tâches donne de meilleurs résultats. Cela ne provient pas vraiment du fait que l'on utilise la migration de tâches mais provient plutôt du partitionnement de tâches. Avec la migration de données, le partitionnement s'effectue jusqu'aux éléments du vecteur, alors qu'avec la migration de tâches, le partitionnement s'arrête à un bloc d'éléments.

Avec le partitionnement limité à un bloc, la granularité des tâches est toujours grosse, mais le partitionnement est ralenti par le concept de propriétaire de tâche. Le partitionnement jusqu'aux éléments se fait plus rapidement car le nombre de tâches est plus élevé et n'importe quelle station peut exécuter n'importe quelle tâche. Cependant, les résultats avec ce dernier sont moins bons à cause de la granularité de ses tâches. Les premières tâches ont une grosse granularité car elles correspondent à de grosses portions du tableau, mais au fur et à mesure que le programme avance, la granularité des tâches

diminue, ce qui réduit les performances. Ainsi, la lenteur du partitionnement limité à des blocs est compensée par la grosseur de la granularité des tâches.

On peut donc voir qu'il est préférable de limiter le partitionnement à des blocs plutôt qu'aux éléments. Cependant, comme cela complexifie l'algorithme et qu'un des buts de ParSubC est d'offrir un modèle de programmation simple, nous avons préféré présenter les résultats du programme le plus simple. Pour ce qui est de la version avec migration de tâches, le partitionnement en bloc est nécessaire pour obtenir des accélérations.

6.4 Discussion sur les implantations

Maintenant que les résultats des programmes ParSubC ont été présentés pour les deux architectures, il est temps de comparer ces résultats. Nous pourrons comparer l'impact des architectures et des implantations sur les performances du langage ParSubC. D'autres mesures sont aussi présentées pour approfondir la discussion, mais à cause d'un problème d'accès au Cray T3D, certaines mesures ne sont disponibles que pour le réseau de stations de travail HP.

Comme il a été mentionné dans la section précédente, et comme en témoignent les données présentées aux tables 6.1 et 6.2, le Cray T3D offre de bien meilleures performances que notre réseau de stations de travail. Il y a plusieurs facteurs qui influencent les résultats obtenus, tant au niveau de l'architecture que de l'implantation. Pour faciliter la discussion, nous aborderons ces facteurs sous deux angles différents, soit le partitionnement et la mémoire partagée.

6.4.1 Partitionnement de tâches

Les résultats obtenus avec le partitionnement de ParSubC, soit l'opérateur `!!` basé sur la CPT, sont assez intéressants. L'exécution des programmes `fib` et `queens` sur le Cray donne des accélérations presque linéaires. Comme ces programmes ne font aucunement accès à la mémoire partagée, ils nous permettent de bien voir le partitionnement de tâches à l'oeuvre. Les résultats obtenus avec ces deux mêmes programmes sur les stations de travail peuvent sembler moyens (l'accélération est de moins de 6 avec 16 processeurs), mais lorsqu'on regarde les profils d'exécution de l'annexe B, on peut voir que le parallélisme disponible est intéressant, et que les tâches sont bien partitionnées. Dans les deux cas, après environ 100 ms, le travail s'est réparti sur presque tous les processeurs. Mais comme le temps d'exécution total des programmes est plutôt petit,

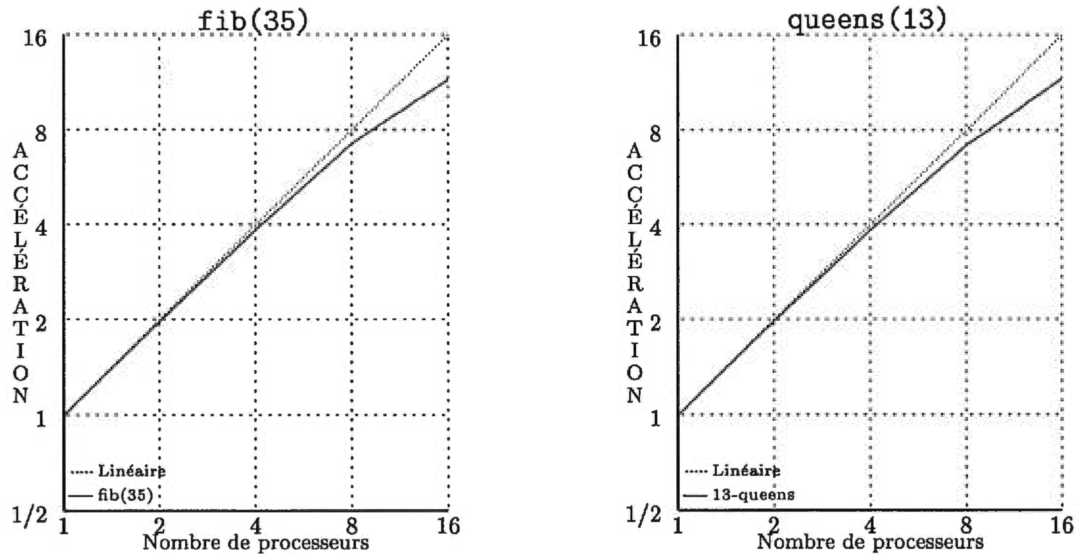

FIG. 6.5 - `queens` *et* `fib` *avec granularité plus élevée sur le multi-ordinateur.*

pas plus de $\frac{1}{2}$ seconde, les processeurs ont à peine commencé l'exécution parallèle des tâches, que déjà le programme est en voie de se terminer. À ce point, les retours de vols de tâches prennent place et le nombre de tâches devient insuffisant. Le réseau étant un moyen de communication plutôt lent, le temps de partitionnement initial pour 16 processeurs et celui des retours de vols de tâches ont un impact trop élevé sur un programme de $\frac{1}{2}$ seconde pour avoir une bonne accélération. Deux exemples qui viennent appuyer notre explication sont les accélérations de `fib` et `queens` avec une plus grosse granularité présentées dans la figure 6.5. Les profils d'exécution de ces deux programmes disponibles dans l'annexe B montrent une bien meilleure utilisation des ressources.

Influence de la migration de tâches

La migration de tâches permet de diminuer le nombre de migrations de données des programmes ayant une fréquence élevée d'écritures à des variables partagées. Cependant, la programmation basée sur la migration de tâches introduit le concept de propriétaire de données, ce qui peut influencer grandement le partitionnement des tâches. Le problème est que ce concept se répercute sur les tâches faisant référence à ces données. Ainsi une tâche faisant référence à une telle donnée ne peut être complétement exécutée que par le propriétaire de la donnée.

Comme le concept de *propriétaire de tâche* ne se retrouve pas avec les vols de tâches,

Programmes	T_{Seq}	$T_1^{Sans!!}$	T_1	Exposition du parallélisme	
				G	P
`queens`	—	1.86	2.05	1.10	—
`poly`	—	1.75	1.75	1.00	—
`sum`	0.15	0.55	0.64	1.16	3.67
`fib`	0.01	1.12	1.33	1.19	—
`mm`	1.75	2.12	2.21	1.04	1.21
`abisort`	—	3.28	3.37	1.03	—
`scan`	0.20	1.20	1.22	1.02	6.00

TAB. 6.5 - *Mesures sur le partitionnement.*

lorsqu'une station reçoit une requête de vol, il retournera la tâche du dessous de sa pile sans égard aux données référencées par celle-ci. Le voleur devra commencer l'exécution de la tâche pour se rendre compte qu'il doit la migrer. Le problème n'est pas tellement que la migration de tâches soit si coûteuse (seulement deux envois de messages), mais le voleur se retrouve maintenant sans travail et doit se remettre à la recherche de travail. Avec la migration de tâches, le vol de tâche n'est donc plus une méthode de partitionnement aussi efficace puisque certaines tâches ne peuvent être exécutées que par une seule station.

Gestion des tâches

Avec les données T_1 et $T_1^{Sans!!}$ de la table 6.5, il a été possible de calculer le coût G associé à l'empilement et au dépilement des tâches. À cause de difficultés d'accès au Cray T3D, les mesures présentées dans cette table ne concernent que le réseau de stations HP. La discussion de la présente section et de celles qui suivent ne concernent donc que l'implantation pour le multi-ordinateur.

Dans cette table, on peut voir que le coût pour l'empilement et le dépilement varie entre 1.00 et 1.19 dépendant des problèmes. Un coût proche de 1 indique que le temps associé à la gestion des tâches n'est pas significatif par rapport au temps total d'exécution. C'est le cas de `poly` qui donne un temps de 1.75 secondes pour les deux exécutions. À l'opposé, il y a le programme `fib` où la gestion des tâches ajoute 19 pour cent au temps d'exécution du programme. Avec `fib`, initialement les tâches sont assez grosses, mais elles sont divisées en sous-tâches, qui à la fin de l'arbre, ont une granularité très fine. Ainsi pour `fib`, le coût de gestion des tâches est plus significatif par rapport au temps d'exécution total que pour les autres programmes. Le surcoût moyen est de

	Facteurs de dérangement			
Programmes	F_2	F_4	F_8	F_{16}
`queens`	1.77	1.54	1.49	1.33
`poly`	1.84	1.54	1.49	1.35
`sum`	1.73	1.61	1.55	1.21
`fib`	1.68	1.52	1.43	1.41
Moyennes	1.76	1.55	1.49	1.33

TAB. 6.6 - *Facteur de dérangement des stations libres.*

8 pour cent, ce qui peut être facilement compensé par le parallélisme offert dans les programmes.

6.4.2 Facteur de dérangement des processeurs libres

Pour évaluer quel impact ont les processeurs libres sur l'exécution d'un programme parallèle ParSubC, certains programmes ont été compilés sans l'opérateur `!!`, mais ont quand même été exécutés sur 1 à 16 processeurs. Cela nous permettra de voir quelle charge les processeurs libres ajoutent à un processeur qui travaille. Le problème dans des situations comme celles-ci, où le degré de parallélisme est inférieur au nombre de processeurs, est qu'il y a des processeurs libres et ceux-ci passent leur temps à envoyer des requêtes de vol, ce qui encombre les processeurs qui travaillent.

Avec les données de la table 6.6, on peut voir que pour tous les programmes, le facteur de dérangement est assez important avec deux processeurs, mais que heureusement ce facteur diminue lorsque le nombre de processeurs augmente. Cela est probablement du au fait que les processeurs libres se dérangent entre eux, ce qui fait diminuer le nombre de requêtes de vol destinées au processeur travaillant. On peut aussi voir que ce facteur de dérangement est assez semblable d'un programme à l'autre.

Bien que le facteur diminue lorsque le nombre de processeurs augmente, il n'en demeure pas moins que si à un moment dans un programme, le degré de parallélisme devient inférieur au nombre de processeurs, l'impact n'est pas négligeable. Le phénomène de dérangement des processeurs libres est une source potentielle d'inefficacité qu'il vaudrait la peine d'attaquer.

Il est intéressant de noter que ce problème n'est pas présent dans l'implantation de ParSubC pour le Cray T3D. Avec cette implantation, avant d'envoyer une requête de vol,

Programmes	Coût du "polling" (en pourcentage)
queens	24.4
poly	20.6
mm	12.8
sum	9.9
fib	9.4
scan	8.2
abisort	5.1
Moyenne	12.9

TAB. 6.7 - *Impact du "polling" dans l'implantation avec mémoire partagée*

un processeur vérifie toujours si sa victime potentielle possède des tâches dans sa pile. La mémoire partagée du Cray T3D a donc permis d'éviter le problème de dérangement des processeurs libres.

6.4.3 Coût du "polling" avec le Cray T3D

Un point particulier de l'implantation avec mémoire partagée qu'il serait intéressant d'analyser est l'impact des points de "polling". L'avantage que possède cette technique par rapport aux interruptions matérielles est d'avoir un temps de traitement plus petit. En contre-partie, le "polling" ajoute un coût aux programmes ParSubC à cause des points de "polling" répartis à travers les programmes. Pour évaluer ce coût, des versions sans point de "polling" ont été exécutées et les temps d'exécution ont été comparés à ceux des versions avec "polling". Les coûts associés au "polling" sont présentés en pourcentage dans la table 6.7.

L'impact du "polling" n'est donc pas négligeable, puisqu'il varie entre 5.1 et 24.4 pour cent pour une moyenne de 12.9 pour cent. Avec une exécution séquentielle, les interruptions matérielles auraient été plus efficaces puisqu'elles n'ajoutent aucun coût lorsqu'il n'y a pas d'interruption. C'est avec un nombre plus élevé de processeurs que le "polling" peut devenir plus intéressant, car la force du "polling" est sa rapidité de traitement des requêtes. Plus le nombre de processeurs augmente, plus la fréquence de requêtes à traiter augmente, et cela favorise donc de plus en plus le "polling".

Il aurait été intéressant de mesurer les performances d'une implantation pour le Cray T3D basée sur les interruptions matérielles pour pouvoir quantifier le "polling"

Accès	Temps en *ms* (en fonction du nombre proc.)			
	2	4	8	16
Référence	2.16	—	—	—
Affectation	0.194	—	—	—
Diffusion	0.221	0.736	2.059	4.055

TAB. 6.8 - *Temps d'accès à la MPV.*

face aux interruptions matérielles, mais cela dépasse la portée de ce travail puisque cette implantation n'est que secondaire et ne sert qu'à titre comparatif.

6.4.4 Mémoire partagée

Un élément qui peut avoir beaucoup d'impact sur les accélérations est la mémoire partagée, et ce autant pour le Cray que pour le réseau de stations. Par exemple, les temps d'exécution des programmes `poly` et `mm` sur le Cray, sont augmentés à cause de la mémoire de type NUMA du Cray. Ainsi, les programmes qui reposent beaucoup sur la mémoire partagée ont de la difficulté à obtenir de bonnes accélérations, à cause des nombreux accès au réseau qu'ils requièrent. Ce problème ne provient cependant pas de nos implantations, mais plutôt d'une caractéristique matérielle.

Mémoire partagée virtuelle

Dans le cas du réseau de stations de travail, la mémoire partagée affecte de façon bien plus importante les performances des programmes. Notre MPV étant basée sur le réseau, les temps d'accès sont extrêmement élevés par rapport à ceux d'accès mémoire (voir table 6.8 [1]), et bien que fonctionnelle, notre MPV doit être utilisée avec réserve. Pour les variables globales, cela n'est vrai que pour les affectations, car les références ne sont que de simples références locales. Ainsi, les problèmes comme `scan`, `abisort` et `mm`, qui font des affectations à tous les éléments d'un vecteur global n'arrivent même pas à obtenir une accélération, alors que ceux comme `sum` et `poly` qui travaillent aussi sur des vecteurs globaux, mais qui ne font que des références, donnent de bien meilleurs résultats.

1. Pour calculer les temps d'accès de la MPV, pour chacune des opérations de la MPV, une boucle exécutant 1000 fois l'opération de MPV a été exécutée, puis le temps moyen de l'opération a été calculé en divisant le temps d'exécution de la boucle par 1000.

Un résultat particulièrement intéressant à analyser face à la mémoire partagée est celui du programme `poly`. Les accélérations obtenues sont assez semblables pour les deux architectures (avec 16 processeurs, l'accélération est de 6.08 pour le Cray et de 5.42 pour le réseau de stations). C'est le seul programme où les résultats sont si semblables, mais ce n'est cependant qu'une coïncidence. Si les résultats sont si bas pour le Cray, c'est à cause des nombreuses références au vecteur qui est global, et requiert donc des accès au réseau d'interconnexion. Dans le cas du réseau de stations, si les résultats sont supérieurs à la moyenne, c'est que malgré que ce problème travaille sur un vecteur global, les accès ne sont que des références, et donc que des accès locaux.

Il serait intéressant de se demander que seraient les performances des programmes travaillant sur des vecteurs globaux si le modèle retenu pour les MPV était d'avoir une seule copie des variables globales (comme c'est le cas pour le Cray). Cela aurait pour effet de diminuer grandement le temps d'affectation à des variables globales (surtout avec un nombre de processeurs élevé), mais augmenterait le temps d'une référence, puisqu'un accès au réseau serait maintenant nécessaire. Évidemment les programmes faisant beaucoup et principalement des diffusions auraient de bien meilleures performances, et ceux basés surtout sur les références verraient leur temps d'exécution augmenter. Ainsi il n'y a pas de modèle clairement supérieur à un autre, puisque cela dépend des programmes. Mais de façon générale, le problème de base reste le même: les programmes provoquant beaucoup d'accès au réseau donneront des performances faibles.

En conclusion, la MPV est fonctionnelle et tout programme ParSubC nécessitant une mémoire partagée pourra être exécuté avec la MPV. Cependant, si on garde en tête que le but du parallélisme est d'améliorer les performances, la MPV perd beaucoup de son attrait et doit être utilisée avec réserve.

6.4.5 Coût de migration d'une tâche

Pour calculer le coût de migration d'une tâche, nous n'avons pas mesuré l'exécution de programmes basés sur la migration de tâches. Nous avons plutôt, connaissant l'implantation pour réseau de stations, estimé ce coût à l'aide des mesures prises pour la MPV (voir la table 6.8). Migrer une tâche consiste à faire une affectation à la pile de tâches prêtes d'une station et se situe donc aux alentours de 0.2 ms.

6.4.6 Surcoût associé à l'accès aux variables partagées

	Temps d'exécution		
Programmes	Sans	Avec	Facteur
`abisort`	0.66	3.37	5.1
`scan`	0.36	1.22	3.4
`square`	0.26	0.58	2.2
`sum_mt`	0.16	0.27	1.7
`mm`	1.57	2.25	1.4
`scan_mt`	1.47	1.68	1.1

TAB. 6.9 - *Surcoût associé aux variables partagées*

Jusqu'à présent nous avons mesuré les coûts associées à l'accès à la MPV et à la migration de tâches lors d'utilisation de pointeurs annotés `light` ou `heavy`, mais nous n'avons pas mesuré l'impact des vérifications associées à ces accès (c'est-à-dire déterminer si une variable est globale ou locale, et dans ce dernier cas, qui est en le propriétaire). Dans le cas où l'accès résulte en un accès à la MPV ou à la migration d'une tâche, ce surcoût est très négligeable, mais lorsque ce n'est pas le cas, lors d'une exécution séquentielle par exemple, il serait intéressant de savoir quel impact ont ces vérifications.

De plus, les programmes dont l'algorithme repose fortement sur l'écriture dans des variables globales ont également un surcoût associé à l'utilisation de variables partagées, car les valeurs des variables globales doivent être diffusées.

Pour mesurer l'impact de tout cela, nous avons modifié les programmes utilisant l'annotation `light` ou `heavy` ainsi que ceux dont l'algorithme repose sur l'écriture dans des variables globales, en leur enlevant les vérifications et les diffusions, pour ensuite les exécuter avec une seule station. Les résultats obtenus sont présentés dans la table 6.9 en plus des résultats des exécutions avec vérifications et diffusions. On peut voir que ces opérations ont un surcoût qui varie entre 1.1 et 5.1.

Malgré que certains programmes ont un surcôut assez élevé, cela n'a pas vraiment d'impact avec des exécutions parralèles, car le surcoût demeure toujours négligeable par rapport au coût des accès à la MPV ou à celui de migrer une tâche.

6.4.7 Code généré

À titre informatif, nous avons inclus dans ce document le coût associé à la compilation d'un programme ParSubC. Il faut se rappeler que les compilateurs ParSubC

Programmes	Compilateurs		Facteur
	C natif	ParSubC	
poly	0.41	1.74	4.2
sum_mt	0.04	0.16	4.0
mm	0.60	1.55	2.6
fib	0.55	1.09	2.0
queens	0.98	1.85	1.9
abisort	0.34	0.61	1.8
sum	0.30	0.55	1.8
scan	0.20	0.32	1.6
square	0.17	0.26	1.5
scan_mt	1.27	1.45	1.1

TAB. 6.10 - *Compilateur ParSubC versus C.*

génèrent du code C, qui est ensuite compilé par un compilateur C. Les données de la table 6.10 permettent donc de voir le coût associé à la génération du code C intermédiaire fait pour les programmes ParSubC. On peut voir que pour les programmes utilisés dans ce travail, les programmes compilés avec ParSubC sont de 1.1 à 4.2 fois plus lents que les programmes compilés directement avec un compilateur C.

Malgré que le coût du code intermédiaire généré par ParSubC soit élevé, il ne faut pas perdre de vue que peu d'efforts ont été mis à la qualité du code généré. Les compilateurs implantés pour ce travail n'étaient que des prototypes, et les coûts présentés dans la table 6.10 diminueront de beaucoup lorsque le compilateur ParSubC génèrera directement son code machine. En contre-partie, la génération de code C du compilateur ParSubC a offert une grande portabilité.

6.5 Modèle de programmation

Un autre aspect auquel il serait intéressant de s'attarder est le modèle de programmation de ParSubC. Après avoir écrit quelques programmes ParSubC, on peut affirmer que le modèle de style *diviser-pour-régner* est assez général et rend la parallélisation des programmes simple et facile. Pour ce qui est des performances obtenues, elles sont assez bonnes si on ne tient pas compte des problèmes provenant des accès aux mémoires partagées. Étant donné la nature du moyen de communication du multi-ordinateur, l'implantation de ParSubC sur celle-ci exige des tâches d'une granularité supérieure à

celle de l'implantation du Cray pour avoir des résultats semblables. Mais si on ne tient compte que du modèle de programmation lui-même, les deux implantations peuvent offrir des accélérations intéressantes.

Pour vraiment connaître les performances que nous offre notre modèle de programmation, il faut aussi tenir compte des résultats obtenus avec ceux de versions séquentielles optimales des programmes. Pour cela, des versions séquentielles des programmes `fib`, `mm`, `scan` et `sum`, ont été écrites et exécutées, et les résultats des exécutions ont été comparés à ceux des versions parallèles sans `!!` sur 1 processeur. Les comparaisons sont quantifiées par la variable P présentée dans la table 6.5, et mesurent le coût provenant de la mise sous forme parallèle des programmes.

Les valeurs de P pour les programmes `mm`, `scan` et `sum` varient entre 1.21 et 6. Ces valeurs sont reliées principalement au surcoût de la mise sous forme diviser-pour-régner des programmes. Ainsi, alors que les programmes séquentiels sont constitués de boucles `for`, celles-ci sont transformées en appels de fonction dans les versions séquentielles. Le premier impact de cette transformation est qu'un appel de fonction est plus lent à exécuter qu'un simple tour de boucle. Ensuite, pour une boucle de 1 à n, il y aura $2*n$ appels de fonction, car le partitionnement diviser-pour-régner se fait selon un arbre (contenant n feuilles) et, à chaque noeud et feuille de l'arbre, il y a un appel de fonction. Regardons de plus près, la valeur de P de chacun des programmes:

- `mm` – La valeur de P de ce programme est de 1.21, ce qui est petit. La principale raison est que ce programme est constitué de trois boucles imbriquées, mais que seulement les deux premières boucles ont été mises sous forme diviser-pour-régner. La granularité du travail fait aux feuilles (qui consiste donc en une boucle de 1 à n) est assez élevée pour rendre presque négligeable le surcoût des appels récursifs requis pour s'y rendre.
- `sum` – La valeur de P de 3.67 de ce programme montre bien le surcoût associée à la mise sous forme diviser-pour-régner d'une boucle. La granularité du travail aux feuilles et aux noeuds étant très petite, la gestion des appels de fonction constitue une grande part du temps d'exécution. Cette valeur est quand même acceptable considérant le parallélisme offert par l'algorithme diviser-pour-régner.
- `scan` – La valeur de P de 6.00 de ce programme s'explique, comme pour `sum`, par la faible granularité du travail fait aux noeuds et aux feuilles de l'arbre. Mais en plus, comme l'algorithme utilisé pour `scan` nécessite deux passes, sa valeur de P est presque le double de celle de `sum`. Si elle ne l'est pas tout à fait, c'est que la granularité d'une itération de `scan` est plus élevée que celle de `sum`, ce qui rend un

peu plus négligeable le surcoût de la transformation des boucles en forme diviser-pour-régner. Même si la valeur de P du programme `scan` est plus élevée que celles de `sum` et `mm`, elle peut quand même être compensée par le parallélisme offert dans la version parallèle.

Pour ce qui est de la valeur de P du programme `fib`, nous n'avons pas utilisé comme version séquentielle celle qui consiste en une simple boucle. La valeur de P de celle-ci, qui serait alors de de 112, n'est pas très indicative, car l'intérêt de paralléliser la version doublement récursive n'est pas son efficacité mais plutôt la présentation du langage ParSubC. Le problème de la suite de Fibonacci pouvant se résoudre très efficacement pour de petites valeurs de n, la parallélisation de ce programme ne serait intéressante, au point de vue performance, que pour de grandes valeurs de n. Dans de tels cas, l'algorithme parallèle utilisé ne serait sûrement pas celui doublement récursif, mais plutôt une mise sous forme diviser-pour-régner de la boucle séquentielle (comme pour `sum` et `scan`). L'algorithme doublement récusif n'a été uniquement utilisé que pour présenter le langage ParSubC.

6.5.1 Influence de la MPV

Bien que le modèle de programmation avec l'opérateur `!!` soit simple, la MPV complique la programmation avec ParSubC sur le multi-ordinateur. Le problème est que le parallélisme ayant pour but la performance des programmes, et la MPV offrant un support de mémoire partagée très inefficace, un programmeur se verra quelquefois forcé de prendre conscience des accès au réseau présents dans ses programmes, s'il veut obtenir de bons résultats. Un exemple qui illustre bien ce problème est l'initialisation des vecteurs globaux de certains programmes. La méthode naturelle qui consiste à initialiser chaque élément du vecteur est à éviter, car cela provoquera une diffusion pour chacun des éléments du tableau. Une méthode beaucoup efficace consiste à déclarer le tableau dans une structure, d'initialiser un tableau similaire mais local, puis de diffuser le tableau d'un seul coup en affectant le tableau local au tableau global. L'utilisation d'une structure était requise, car il n'est pas possible de faire l'affectation d'un tableau au complet en C. Cette contorsion alourdit l'écriture des programmes ParSubC, mais est requise pour éviter le temps excessif provenant de la première méthode.

Un autre exemple provient du programme `mm`. La version qui a été utilisée est la même que pour le Cray, et est celle présentée dans l'annexe A. Avec cette version, l'opérateur `!!` est appliqué une première fois pour que les rangées soient calculées en parallèle, puis une deuxième fois pour que tous les éléments d'une même rangée puissent être calculés

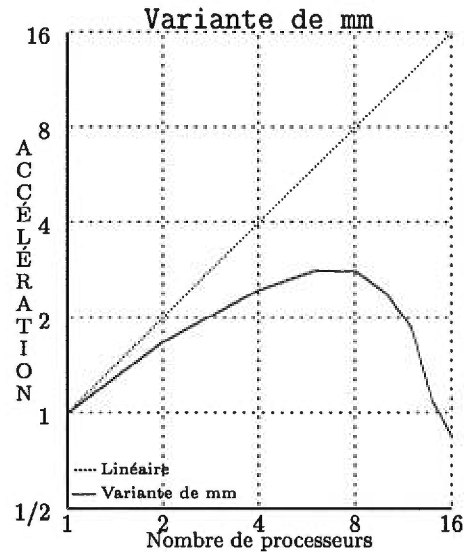

FIG. 6.6 - *Variante de* mm *pour diminuer le nombre de diffusions.*

également en parallèle. Cependant, cet algorithme a le désavantage de nécessiter une diffusion pour chaque élément, soit un total de 22500 diffusions (150 × 150), ce qui cause un engorgement du réseau. À cause de la lenteur de la MPV, cet algorithme n'est vraiment pas adéquat pour le multi-ordinateur.

Afin de diminuer le nombre de diffusions, une variante du programme mm a été écrite (voir le programme `mm-variante` dans l'annexe A). Plutôt que d'appliquer l'opérateur !! une deuxième fois pour calculer les éléments d'une rangée en parallèle, ceux-ci sont calculés séquentiellement. Il y a donc moins de tâches, mais de cette façon, il est possible de diffuser une rangée complète, avec le truc décrit précédemment, plutôt que de diffuser les éléments un par un. Cela diminue donc le nombre de diffusions à 150, ce qui est beaucoup plus raisonnable. Cette prise de conscience des accès au réseau n'est pas négligeable puisqu'elle permet d'avoir une variante de mm qui offre maintenant des accélérations (voir figure 6.6). Les accélérations sont loin d'être linéaires mais c'est déjà beaucoup mieux que la version initiale de mm. Le cas de mm est un bel exemple où la MPV vient influencer, pour ne pas dire compliquer, le modèle de programmation de ParSubC.

6.5.2 Migration de tâches

À cause de la lenteur des accès à la MPV, le préfixe `heavy` a été mis à la disposition du programmeur pour permettre de diminuer le nombre de migrations de données. La migration de tâches introduit cependant dans ParSubC le concept de partitionnement de données. Par exemple, avec les programmes présentés dans ce chapitre, le vecteur est initialement explicitement distribué à travers les stations. Le modèle de programmation basée sur la migration de tâches est donc plus complexe que celui basée sur la migration de données. Ce partitionnement des données que l'on retrouve dans la programmation avec migration de tâches lui donne un style qui s'apparente au parallélisme de données.

De plus, avec les inconvénients rencontrés par rapport au partitionnement de tâches, la migration de tâches complexifie le modèle de programmation encore bien plus que la MPV ne l'a fait. La migration de tâches donne à ParSubC une saveur de *langage pour expert* puisqu'il faut connaître certains détails techniques pour arriver à écrire de bons programmes. Par exemple, il faut bien analyser comment se feront le partition et la migration des tâches si on veut obtenir de bons résultats. Il ne faut cependant pas perdre de vue que le préfixe `heavy` permettant la migration de tâches est présent dans ParSubC uniquement pour des raisons de performances, soit pour contrer la lenteur de la MPV. Tout programme écrit avec `heavy` peut facilement se convertir avec `light`, et la seule motivation pour utiliser l'annotation `heavy` plutôt que `light` est pour obtenir de meilleures accélérations.

Il serait cependant intéressant de se demander ce qu'auraient été les résultats si la migration de tâches avait été implantée de façon à migrer la tâche courante plutôt que le dernier appel de fonction. Cela aurait-il pu simplifier le modèle de programmation?

Après une brève analyse, il semble probable que le nombre de tâches migrées soit beaucoup plus petit puisqu'à la dernière récursivité, la tâche courante contenant tous les appels de fonctions imbriqués serait migrée, ce qui éviterait d'avoir à migrer chacun des appels de fonction. Cependant, le problème du dépassement de la taille des structures de la CPT serait probablement encore présent car, comme il a été présenté dans la section 4.6.1, les tâches légères ainsi que la tâche courante doivent être conservées dans les piles de la première station même après la migration. Le problème avec cette technique est donc qu'une même tâche peut occuper les piles de plusieurs stations.

6.6 Résumé

Pour évaluer les implantations de ParSubC, plusieurs programmes ParSubC ont été écrits et les temps d'exécution sur le Cray T3D et le réseau de stations de travail HP ont été mesurés et comparés. Des versions séquentielles ont également été écrites pour évaluer le coût de parallélisation d'un programme avec ParSubC. Avec ces résultats, les points suivants ont pu être démontrés:

- Le coût ajouté pour l'exposition du parallélisme est raisonnable et peut être compensé par le parallélisme disponible dans les programmes ParSubC. Un premier facteur, soit le coût de gestion de tâches ajouté par l'opérateur `!!`, ajoute au pire 20 pour cent au temps d'exécution. La mise sous forme *diviser-pour-régner* des programmes n'a pas ajouté un coût excessif ne pouvant être compensé par le parallélisme. Cependant, ce deuxième facteur est plus élevé que le premier (entre 1.16 et 5.88 fois plus élevé).
- L'architecture du Cray T3D étant plus performante que celle du multi-ordinateur au niveau de la mémoire partagée, l'implantation de ParSubC pour ce dernier offre de moins bons résultats que celle pour le Cray. La différence majeure provient de la mémoire partagée du Cray T3D, qui offre un moyen de communication interprocesseur rapide qui permet au Cray:
  - d'avoir un partitionnement plus efficace.
  - d'offrir un support adéquat pour les programmes nécessitant une mémoire partagée.
  - d'éviter que les processeurs libres dérangent ceux qui travaillent lorsque ceux-ci n'ont pas de tâches.
- La MPV est fonctionnelle mais doit être utilisée avec réserve, car pour certains usages, sa performance est mauvaise. Pour contrer sa lenteur, le programmeur peut partitionner ces données à travers les stations et avoir recours à l'annotation `heavy`. La migration de tâches offre de meilleures performances sur les programmes qui ont une fréquence élevée d'écritures sur des variables partagées. Pour avoir de bons résultats, ces programmes doivent cependant travailler (écriture et lecture) localement sur les variables partagées. Le modèle de programmation avec l'annotation `heavy` est cependant plus complexe que celui avec `light`.
- De façon générale, le modèle de programmation de ParSubC est simple et peut offrir de bonnes performances (si on ne tient pas compte des limites imposées par les architectures).

Chapitre 7

Conclusion

Ce travail avait comme but de définir un nouveau langage parallèle basé sur le langage C. De tels langages existent déjà, mais très peu sont destinés au parallélisme de contrôle, et ceux qui le sont, ont parfois une syntaxe assez lourde, ou encore, leur modèle de programmation n'est pas assez souple. Notre nouveau langage, qui n'est en réalité qu'une petite extension au langage C, a été défini pour offrir un modèle de programmation simple, souple et portable. Contrairement à d'autres langages parallèles, ParSubC ne requiert que peu de connaissance du parallélisme pour être utilisé.

Pour pouvoir évaluer la qualité du langage ParSubC, deux compilateurs ont été écrits pour deux architectures parallèles différentes. La première, qui consiste en un réseau de stations de travail HP, a l'avantage d'être une architecture très disponible, mais n'offre pas le meilleur support comme architecture parallèle. L'autre architecture, le Cray T3D, est un ordinateur multi-processeur avec mémoire partagée spécialement conçu pour le parallélisme.

Ces deux implantations de ParSubC ont permis d'évaluer les performances de programmes écrits avec ParSubC. Les résultats ont principalement démontré que:

- Le partitionnement, basé sur la CPT, permet aux programmes parallélisés avec le langage ParSubC, d'obtenir de bonnes performances, et le coût ajouté pour la parallélisation sous forme *diviser-pour-régner* d'un programme est raisonnable (par rapport à la version séquentielle).
- Le multi-ordinateur offre un moins bon support pour ParSubC, et par conséquent, les performances obtenues sont moins bonnes. Le partitionnement permet quand

même d'obtenir de bons résultats, mais la granularité des tâches doit être plus élevée pour obtenir des résultats semblables à ceux du Cray T3D.

- La mémoire partagée virtuelle de notre multi-ordinateur est beaucoup trop lente pour exécuter efficacement des programmes reposant fortement sur une mémoire partagée. Bien que fonctionnelle, la mémoire partagée virtuelle doit être utilisée avec réserve. La migration de tâches permet d'améliorer les performances de certains programmes ayant une fréquence d'accès à des variables partagées trop élevée pour avoir de bons résultats avec la MPV. Cependant, la migration de tâches complexifie le modèle de programmation de ParSubC, ce qui lui donne une saveur de *langage pour expert.*

- ParSubC est un langage parallèle simple capable d'offrir de bonnes performances lorsque l'architecture parallèle utilisée lui fournit un support adéquat.

Annexe A

Programmes ParSubC utilisés

Cette annexe contient tous les programmes utilisés pour l'évaluation du langage ParSubC et de ses implantations. Ces programmes sont tous des traductions ParSubC de programmes originalement écrits en Multilisp, et ayant servi à l'évaluation d'une implantation de Multilisp basée sur la création paresseuse de tâches [Fee93]. On retrouve également dans cette annexe, une version séquentielle efficace pour les programmes `poly`, `queens` et `sum`. Pour terminer, on retrouve les programmes ParSubC `sum`, `scan` et `square` basés sur la migration de tâches. Une brève description est également fournie avec chaque programme dans les sections qui suivent.

A.1 Versions parallèles

A.1.1 abisort

Ce programme fait le tri d'un vecteur de 32768 éléments en utilisant l'algorithme de tri bitonique tel que décrit dans [BN89].

```
/* abisort */
#include "header.h"

/* N = 2^K */
#define K 15
#define N 32768

#define swap_value(p1, p2) \
     {int t; t=p1->value; p1->value=p2->value; p2->value=t;}
#define swap_right(p1, p2) \
     {Node*t; t=p1->right; p1->right=p2->right; p2->right=t;}
#define swap_left(p1, p2) \
     {Node *t; t=p1->left; p1->left=p2->left; p2->left=t;}
#define test_and_swap(p1, p2) \
     if(p1->value>p2->value) { swap_value(p1,p2); }
#define new_node(l, r, v, x, T, n) \
     {(T)[*n].right=r; (T)[*n].left=l; (T)[*n].value=v; x=&(T)[(*n)++];}

typedef struct node Node;
struct node
{
  Node *right, *left;
  int value;
};

struct nodes {Node T[N];};
typedef struct nodes Nodes;
Nodes T;
```

```
void bimerge (light Node *root, light Node *spare)
{
  light Node *pl, *pr;
  int rightexchange, elementexchange;

  rightexchange = (root->value > spare->value);
  if (rightexchange) swap_value(root, spare);
  pl = root->left;
  pr = root->right;

  while(pl != NULL)
  {
    elementexchange = (pl->value > pr->value);
    if(rightexchange)
      if(elementexchange)
      {
        swap_value(pl, pr);
        swap_right(pl, pr);
        pl = pl->left;
        pr = pr->left;
      }
      else
      {
        pl = pl->right;
        pr = pr->right;
      }
    else
      if(elementexchange)
      {
        swap_value(pl, pr);
        wap_left(pl, pr);
        pl = pl->right;
        pr = pr->right;
      }
      else
      {
        pl = pl->left;
        pr = pr->left;
      }
  }
```

```
  if(root->left != NULL)
  {
    bimerge(root->left, root) !! { bimerge(root->right, spare); };
  }
}

Node *make_in_order (int i, int depth, int *max, Node *T, int *n)
{
  Node *l_tree, *r_tree, *r;

  if(depth == 1)
  {
    *max = i;
    new_node(NULL, NULL, i, r, T, n);
    return(r);
  }
  else
  {
    int l_max, r_max;
    l_tree = make_in_order(i, depth-1, &l_max, T, n);
    r_tree = make_in_order(l_max - 2, depth-1, &r_max, T, n);
    *max = r_max;
    new_node(l_tree, r_tree, l_max - 1, r, T, n);
    return(r);
  }
}

void abisort (light Node *root, light Node *spare)
{
  if(root->left == NULL) { test_and_swap(root, spare) }
  else
  {
    abisort(root->left, root) !! { abisort(root->right, spare); };
    bimerge(root, spare);
  }
}
```

```
void main ()
{
  Node *spare, *root;
  int max, n;

  n = 0;
  root = make_in_order(N, K, &max, &T.T[0], &n);
  T = T;
  new_node(NULL, NULL, max-1, spare, T.T, &n);

  begin_time();
  abisort(root, spare);
  end_time("Time to run");
}
```

A.1.2 fib

Ce programme calcule la 30eme valeur de la suite de Fibonacci en utilisant l'algorithme doublement récursif.

```
/* fib */
#include "header.h"

#define N 30    /* Valeur a evaluer */

long int pfib(int n)
{
  if(n < 2)
    return n
  else
  {
    long int f1, f2;

    f1 = pfib(n-1) !! { f2 = pfib(n-2); };
    return f1 + f2;
  }
}

void main()
{
  begin_time();
  pfib(N);
  end_time("Time to run");
}
```

A.1.3 mm

Ce programme fait la multiplication de deux matrices de dimensions 150x150, et mémorise le résultat dans une troisième matrice.

```
/* mm */
#include "header.h"

#define N 150

struct matrice { int m[N][N]; };

struct matrice m1, m2, m3;

void compute_entry (int row, int col)
{
  int sum, k;

  sum = 0;
  for(k=0; k<N; k++)
  {
    sum += m1.m[row][k] * m2.m[k][col];
  }
  m3.m[row][col] = sum;
}

void compute_cols_between (int row, int i, int j)
{
  int mid;

  if(i == j)
    compute_entry(row, i);
  else
  {
    mid = (i+j)/2;
    compute_cols_between(row, i, mid) !!
      { compute_cols_between(row, mid+1, j); };
  }
}
```

```
void compute_rows_between (int i, int j)
{
  int mid;

  if(i == j)
    compute_cols_between(i, 0, N-1);
  else
  {
    mid = (i+j)/2;
    compute_rows_between(i, mid) !! { compute_rows_between(mid+1, j); };
  }
}

void mm()
{
  compute_rows_between(0, N-1);
}

void init_matrices()
{
  int i, j;
  struct matrice *m1p, *m2p;

  m1p = &m1; m2p = &m2;
  for(i=0; i<N; i++)
    for(j=0; j<N; j++)
    {
      m1p->m[i][j] = 2;
      m2p->m[i][j] = 2;
    }
  m1 = *m1p;
  m2 = *m2p;
}
```

```
void main()
{
  init_matrices();

  begin_time();
  mm();
  end_time("Time to run");
}
```

A.1.4 mm-variante

Ce programme est une variante de mm. Cette variante arrête le partitionnement du travail au calcul d'une rangée complète plutôt que de repartitionner la rangée jusqu'au calcul d'un seul élément (comme c'est le cas avec mm). De plus, les rangées sont conservées dans une structure indépendante, ce qui permet la diffusion d'une rangée complète d'un seul coup, plutôt qu'élément par élément.

```
/* mm-variante */
#include "header.h"

#define N 150   /* Nombre de rangees (et de colonnes) des matrices. */

struct vector { int cols[N] };

struct matrice { struct vector rows[N]; };

struct matrice m1, m2, m3;

/*
  On multiplie la rangee avec toutes les colonnes et on conserve le
  resultat.
*/
void mult_row_with_cols (int row)
{
  int col, j, sum;
  vector *p;

  p = &m3.rows[row];
  for(col=0; col<N; col++)
  {
    sum = 0;
    for(j=0; j<N; j++)
      sum += m1.rows[row].cols[j] * m2.rows[j].cols[col];
    p->cols[col] = sum;
  }
  m3.rows[row] = *p;
}

...
```

Le reste de l'algorithme est identique à mm.

A.1.5 poly

Ce programme calcule le carré d'un polynôme de 2000 termes avec coefficients entiers et évalue le polynôme résultant pour un certain x. Ce dernier polynôme n'est pas conservé en mémoire, car l'évaluation avec x se fait au fur et à mesure que le carré est calculé.

```
#include "header.h"
#define N 2000
struct poly { int p[N]; };
struct poly p;

int int_power (int x, int y)
{ int w, res;

  res = 1;
  w = x;
  for(; y != 0; y = y >> 1)
  {
    if( y & 1 ) res = res * w;
    w = w * w;
  }
  return res;
}

int compute_part (int from, int to, int x)
{ int k, i, mini, maxi, sum, T, w;

  T = 0; w = int_power(x, from);
  for(k=from; k<to; k++)
  {
    mini = k-N+1;
    if (mini < 0) mini = 0;
    maxi = k+1;
    if (maxi > N) maxi = N;
    sum  = 0;
    for(i=mini; i<maxi; i++)
      sum += p.p[i]*p.p[k-i];
    T += w*sum;
    w *= x;
  }
  return T;
}
```

```
int eval_poly_square (int i, int j, int x)
{
  int s1, s2, mid;

  if(j - i <= 10)
    return(compute_part(i, j, x));
  else
  {
    mid = (i+j)/2;
    s1 = eval_poly_square(i, mid, x) !!
          { s2 = eval_poly_square(mid, j, x); };
  }

  return (s1+s2);
}

void init_poly()
{
  int i;
  struct poly *pp;

  pp = &p;
  for(i=0; i<N; i++) pp->p[i] = 1;
  p = *pp;
}

void main()
{
  int res;

  res = 0;
  init_poly();

  begin_time();
  res = eval_poly_square(0, 2*n, 1);
  end_time("Time to run");
}
```

A.1.6 queens

Ce programme calcule le nombre de solutions au problème des n-reines avec $n = 12$. Trouver une solution au problème des n-reines consiste à disposer les n reines sur un échiquier de $n \times n$, sans qu'aucune ne puisse en prendre une autre.

```
/* queens */

#include "header.h"

int queens (int i, int diag1, int diag2, int cols, int col)
{
  int free;

  free = diag1 & diag2 & cols;

  if (col > free)
    return 0;
  else
    if (i == 1)
      return 1;
    else
    {
      int s1, s2;
      while (!(col & free)) col = col<<1;
      s1 = queens( i-1, ((diag1-col)<<1)+1,
                  (diag2-col)>>1, cols-col, 1 ) !!
             { s2 = queens( i, diag1, diag2, cols, col<<1 ); };
      return s1+s2;
    }
}

void main()
{
  int n, res;

  begin_time();
  n = 12;
  res = queens( n, -1, -1, (1<<n)-1, 1 );
  end_time("Time to run");
}
```

A.1.7 scan

Ce programme calcule le préfixe parallèle avec la somme d'un vecteur de 131072 éléments et sauve le résultat dans le vecteur existant. Chaque élément est donc remplacé par lui-même, additionné de tous les éléments le précédant.

```
/* scan */
#include "header.h"

#define N 131072

struct vector { int v[N]; };

typedef struct vector Vector;

Vector v;

int pass1 (int i, int j)
{
  int mid, right, left;

  if(i < j)
  {
    mid  = (i+j)/2;

    left = pass1(i, mid) !! { right = pass1(mid+1, j); };
    v.v[j] = left + right;
  }
  return v.v[j];
}

void pass2 (int i, int j, int c)
{
  int mid, cc;

  if(i < j)
  {
    mid = (i+j)/2;
    pass2(i, mid, c) !! { cc = c+v.v[mid]; pass2(mid+1, j, cc); };
    v.v[mid] = cc;
  }
}
```

```
void scan (int c)
{
  pass1(0, N-1);
  pass2(0, N-1, c);
  v.v[N-1] = c+v.v[N-1];
}

void init_v()
{
  int i;
  Vector *vp;

  vp = &v;
  for(i=0; i<N; i++) vp->v[i] = 1 ;
  v = *vp;
}

void main()
{
  init_v();

  begin_time();
  scan(0);
  end_time("Time to run");
}
```

A.1.8 sum

Ce programme calcule tout simplement la somme des éléments d'un vecteur de 500000 éléments.

```
/* sum */
#include "header.h"

#define N 500000

struct vector { int v[N]; };

typedef struct vector Vector;

Vector v;

int sum (int lo, int hi)
{
  int mid, s1, s2;

  if(lo == hi)
    return v.v[lo];
  else
  {
    mid = (lo+hi)/2;
    s1 = sum(lo, mid) !! { s2 = sum(mid+1, hi); };
    return s1+s2;
  }
}

void init_vector()
{
  int i;
  Vector *vp;

  vp = &v;
  for(i=0; i<N; i++) vp->v[i] = 1;
  v = *vp;
}
```

```
void main ()
{
  int res;

  init_vector();

  begin_time();
  res = sum(0, N-1);
  end_time("Time to run");
}
```

A.2 Versions séquentielles

Afin d'évaluer le coût que pouvait engendrer la restructuration sous forme parallèle des programmes, des versions séquentielles ont été écrites pour trois programmes, soit `mm`, `scan` et `sum`. Les versions séquentielles sont toutes basées sur des algorithmes utilisant uniquement des boucles.

A.2.1 mm

L'algorithme utilisé consiste en trois boucles imbriquées.

```
/* mm */
#include "header.h"

#define N 150

int m1[N][N], m2[N][N], m3[N][N];

void mm()
{
  int row, col;

  for(row=0; row<N; row++)
    for(col=0; col<N; col++)
    {
      int *p, k;

      p = &m3[row][col];
      for(k=0; k<N; k++)
        (*p) += m1[row][k] * m2[k][col];
    }
}
```

```
void init_matrices()
{
  int i, j;

  for(i=0; i<N; i++)
    for(j=0; j<N; j++)
    {
      m1[i][j] = 2;
      m2[i][j] = 2;
    }
}

void main()
{
  init_matrices();

  begin_time();
  mm();
  end_time("Time to run");
}
```

A.2.2 scan

L'algorithme utilisé consiste en une simple boucle de 1 à n.

```
/* scan */
#include "header.h"

#define N 131072    /* Taille du vecteur */

int vect[N];

void scan()
{
  int i;

  for(i=1; i<N; i++)
    vect[i] = vect[i] + vect[i-1];
}

void init_vect()
{
  int i;

  for(i=0; i<N; i++)
    vect[i] = 1;
}

void main()
{
  init_vect();

  begin_time();
  scan();
  end_time("Time to run");
}
```

A.2.3 sum

L'algorithme utilisé consiste en une simple boucle de 1 à n.

```
/* sum */
#include "header.h"

#define N 500000        /* Taille du vecteur */

int vect[N];

int sum(int lo, int hi)
{
  int i, s;

  s = vect[lo];
  for(i=lo+1; i<hi; i++)
    s += vect[i];

  return s;
}

void init_vector()
{
  int i;

  for(i=0; i<N; i++)
    vect[i] = 1;
}

void main ()
{
  int res;

  init_vector();

  begin_time();
  res = sum(0, N-1);
  end_time("Time to run");
}
```

A.3 Versions parallèles avec migration de tâches

A.3.1 sum_mt

Ce programme calcule tout simplement la somme des éléments d'un vecteur de 500000 éléments. Le vecteur est divisé en n blocs et chacune des n stations en possède un.

```
/* sum_mt */
#include "header.h"

#define N 500000  /* Taille du vecteur */

heavy char *vect[MAX_NB_NODES];
int block_size;

/*
  Cette fonction divise le travail en deux pour permettre le partitionne-
  ment de taches. Lorsque la taille de l'intervalle est la meme que celle
  d'un bloc,le calcul sur cet intervalle est alors effectue.
*/
int sum(int lo, int hi)
{
  int mid, s1, s2;

  if (hi-lo <= block_size)
  {
    int i, block_id;

    block_id = lo / block_size;
    s1 = 0;
    for(i=lo; i<=hi; i++)
      s1 += vect[block_id][i];
  }
  else
  {
    mid = (lo+hi)/2;
    s1 = sum(lo, mid) II { s2 = sum(mid+1, hi); };
    s1 += s2;
  }
  return s1;
}
```

```
/*
  Initialisation du vecteur  (tous les elements sont de 1).  Le vecteur est
  reparti en n blocs et chaque processeur possede un bloc. L'initialisation
  du vecteur  provoquera des migrations  de taches vers le  proprietaire du
  bloc et l'initiation de chaque bloc se fera localement.
*/
void init_vector()
{
  int i, j, n;

  n = nb_nodes();
  block_size = N / n;
  for(i = 0; i < n; i++)
  {
    vect[i] = shm_malloc_on(i, block_size * sizeof(char));
    for(j = 0; j < block_size; j++)
      vect[i][j] = 1;
  }
}

void main ()
{
  int res;

  init_vector();

  begin_time();
  res = sum(0, N-1);
  end_time("Time to run");
}
```

A.3.2 scan_mt

Ce programme est la version parrallèle avec migration de tâches de scan. Le vecteur est divisé en n blocs et chacune des n stations en possède un.

```
/* scan_mt */
#include "header.h"

#define N 131072    /* Taille du vecteur */

heavy int *vect[MAX_NB_NODES];
int nodes, block_size;

/*
  Le travail est divise en deux pour permettre le partitionnement de
  taches.
*/
int pass1(int i, int j)
{
  int mid, right, left, j_node, j_item;

  j_node = j / block_size; j_item = j % block_size;

  /* Force la migration de tache. */
  mid = vect[j_node][j_item];

  /* Il n'y a qu'un element, on le retourne. */
  if(i == j) return vect[j_node][j_item];

  /* On divise le tableau en deux. */
  mid  = (i+j) / 2;
  left = pass1(i, mid) II { right = pass1(mid+1, j); };

  return (vect[j_node][j_item] = left + right);
}
```

```
/*
  Le travail est divise en deux pour permettre le partitionnement de
  taches.
*/
void pass2(int i, int j, int c)
{
  if(i < j)
  {
    int mid, cc, mid_node, mid_item;

    mid = (i+j) / 2;
    mid_node = mid / block_size; mid_item = mid % block_size;
    cc = c + vect[mid_node][mid_item];
    pass2(i, mid, c) II { pass2(mid+1, j, cc); };
    vect[mid_node][mid_item] = cc;
  }
}

/*
  L'algorithme sequentiel  qui consiste a calculer le premier,  puis le
  2eme, puis le 3eme, ..., n'est pas adequat car il n'y a alors du tra-
  vail que  pour un processeur.  Un algorithme a deux  passes est ainsi
  preferable, puisqu'il offre du parallelisme.
*/
void scan(int c)
{
  pass1(0, N-1);
  pass2(0, N-1, c);
  vect[nodes-1][block_size-1] = c + vect[nodes-1][block_size-1];
}
```

```
/*
  Initialisation du vecteur (tous les elements sont de 1). Pour eviter les
  diffusions, qui sont des operations couteuses, le vecteur est distribue
  dans les memoires locales des stations. Notre vecteur est donc un vecteur
  de pointeurs maintenant.
*/
void init_vector()
{
  int i, j;

  nodes = nb_nodes();
  block_size = N / nodes;
  for(i = 0; i < nodes; i++)
  {
    heavy int *p;
    p = vect[i] = shm_malloc_on(i, block_size * sizeof(int));
    for(j = 0; j < block_size; j++)
      p[j] = 1;
  }
}

void main()
{
  init_vector();

  begin_time();
  scan(N);
  end_time("Time to run");
}
```

A.3.3 square

Ce programme replace chaque élément d'un vecteur par son carré. Le vecteur est divisé en n blocs et chacune des n stations en possède un.

```
/* square */
#include "header.h"

#define N 500000  /* Taille du vecteur */

heavy char *vect[MAX_NB_NODES];
int block_size, nb_blocks;

/*
  Cette fonction divise le travail en deux pour permettre le partitionne-
  ment de taches. Lorsque la taille de l'intervalle est la meme que celle
  d'un bloc, le calcul sur cet intervalle est alors effectue.
*/
void square(int lo, int hi)
{
  if (hi-lo < block_size)
  {
    int i; heavy char *block;

    block = vect[lo / block_size];
    for(i=0; i<block_size; i++)
      block[i] = block[i] * block[i];
  }
  else
  {
    int mid;
    mid = (lo+hi)/2;
    square(lo, mid) II { square(mid+1, hi); };
  }
}
```

```
/*
  Initialisation du vecteur  (tous les elements sont de 3).  Le vecteur est
  reparti en n blocs et chaque processeur possede un bloc. L'initialisation
  du vecteur  provoquera des migrations  de taches vers le  proprietaire du
  bloc et l'initiation de chaque bloc se fera localement.
*/
void init_vector()
{
  int i, j;
  heavy char *block;

  nb_blocks = nb_nodes();
  block_size = N / nb_blocks;
  for(i = 0; i < nb_blocks; i++)
  {
    block = vect[i] = shm_malloc_on(i, block_size * sizeof(char));
    for(j = 0; j < block_size; j++)
      block[j] = 3;
  }
}

void main ()
{
  init_vector();

  begin_time();
  square(0, N-1);
  end_time("Time to run");
}
```

Annexe B

Profils d'exécution

Cette annexe contient les profils d'exécution des programmes ParSubC utilisés pour ce travail sur le Cray T3D et le réseau de stations de travail HP. Les profils d'exécution présentés permettent de visualiser les différentes activités des processeurs à travers le temps. Cela peut être très utile parfois pour comprendre et expliquer les résultats obtenus avec un programme parallèle. Les activités possibles des processeurs sont représentées dans les profils d'exécution par les 5 états suivants:

- **"idle"** — Cet état indique qu'un processeur n'a pas de tâche à exécuter, soit parce qu'il n'a plus de tâches dans sa pile de tâches, soit parce qu'il a du suspendre l'exécution de sa tâche courante. Un processeur dans cet état essaiera de voler une tâche.
- **"working"** — Indique qu'un processeur exécute une tâche. Pour être dans cet état, un processeur ne doit pas seulement avoir une tâche prête à être exécutée, mais il doit vraiment être en train de l'exécuter.
- **"task-operation"** — Cet état correspond au temps passé à la gestion de tâche, soit le traitement d'une requête de vol, le traitement d'une réponse de requête de vol ainsi que le retour d'un vol de tâche. Il est à noter que lorsqu'un processeur envoie une requête de vol, son état est alors **"idle"** et non **"task-operation"**.
- **"broadcasting"** — Indique le temps passé par un processeur pour la diffusion de données globales avec la MPV. Cet état est utile seulement pour le multi-ordinateur.

- **"store-response"** — Lorsqu'un processeur diffuse une donnée globale ou fait une affectation à une variable non-locale, un ou plusieurs processeurs recevront alors un message leur demandant d'affecter telle valeur à telle adresse. Cet état correspond au temps passé à traiter ces demandes d'affectation.

Les profils d'exécution qui suivent sont composés de trois parties. La partie supérieure illustre l'état des processeurs à travers le temps. Le graphique représente, pour chaque état, le pourcentage des processeurs qui sont dans cet état à un moment précis. La partie du milieu indique le pourcentage du temps d'exécution passé dans chaque état. La partie du bas est constituée d'histogrammes illustrant la distribution des durées ainsi que la durée moyenne de chaque état.

Il est à noter que certains profils d'exécution pour le réseau de stations correspondent à une exécution avec 2 processeurs et d'autres avec 16 processeurs. Les programmes qui ont présenté des accélérations avec 16 processeurs ou qui au moins avaient un temps d'exécution raisonnable, sont représentés par un profil d'exécution sur 16 processeurs. Ceux pour qui le temps d'exécution avec 16 processeurs était plus grand qu'avec 2, n'ont qu'un profil d'exécution sur 2 processeurs.

B.1 fib(30)

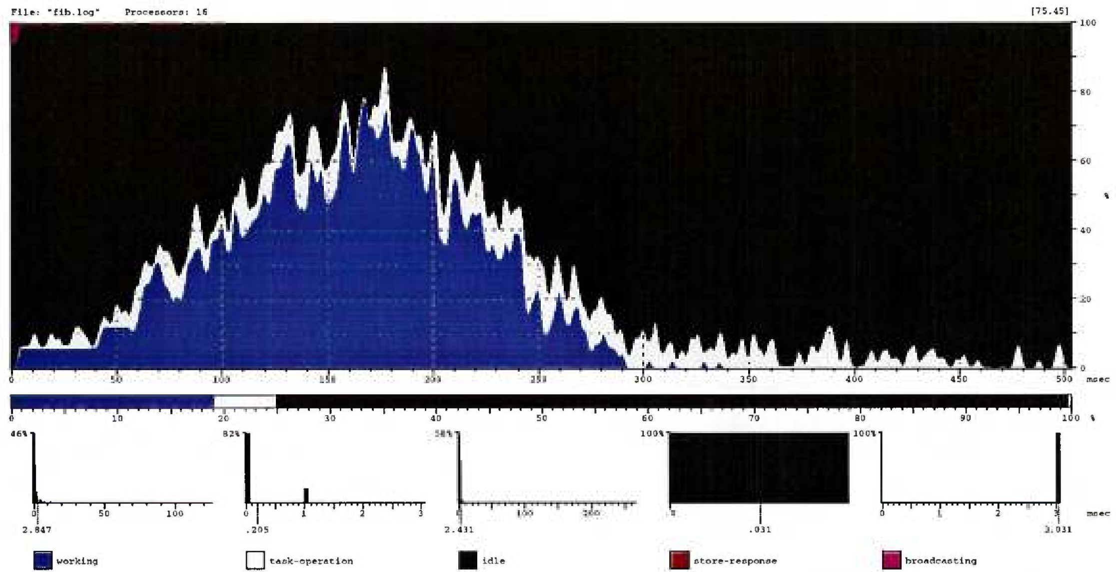

FIG. B.1 - *Profil d'exécution de* `fib30` *sur le multi-ordinateur.*

B.2 fib(35)

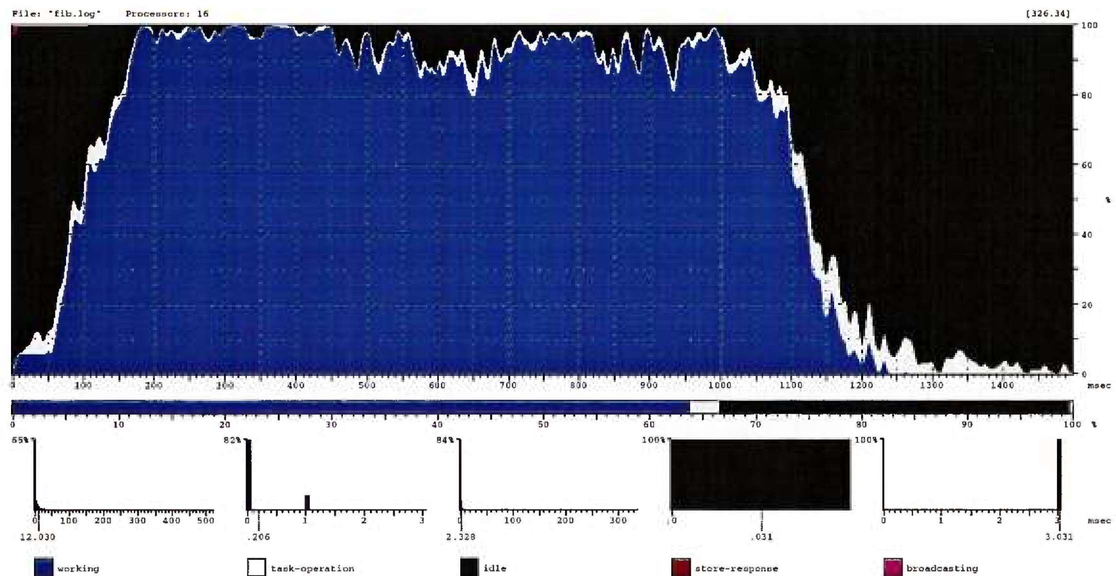

FIG. B.2 - *Profil d'exécution de* `fib35` *sur le multi-ordinateur.*

B.3 12-queens

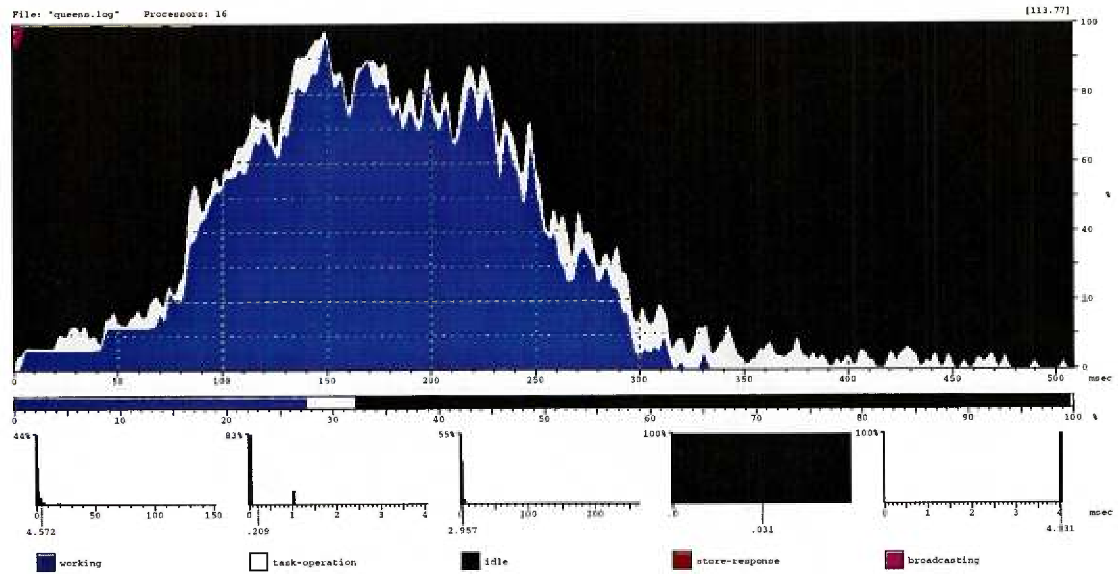

FIG. B.3 - *Profil d'exécution de* 12-queens *sur le multi-ordinateur.*

B.4 13-queens

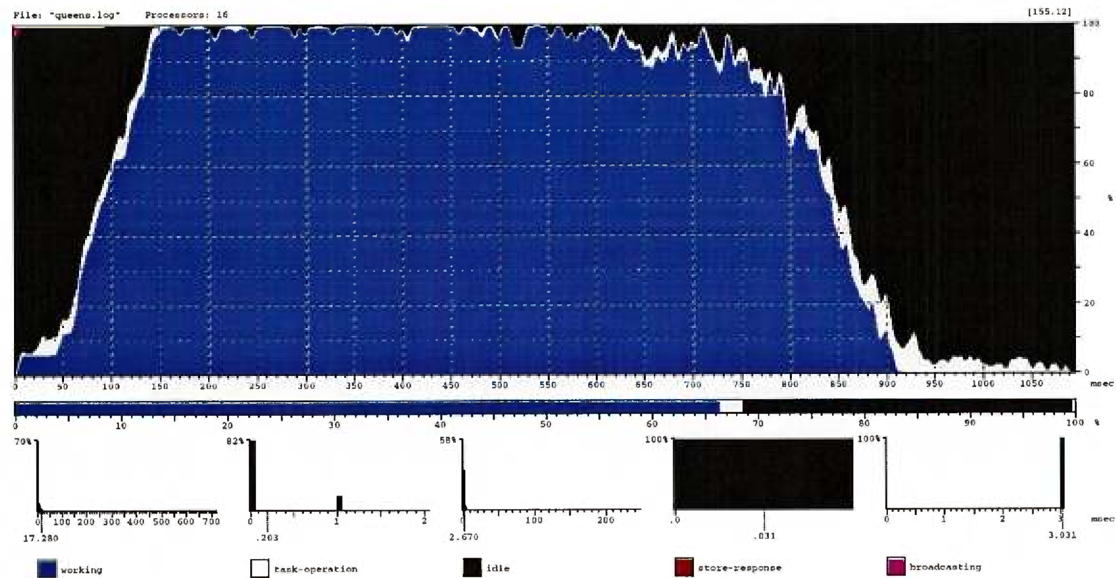

FIG. B.4 - *Profil d'exécution de* 13-queens *sur le multi-ordinateur.*

B.5 mm

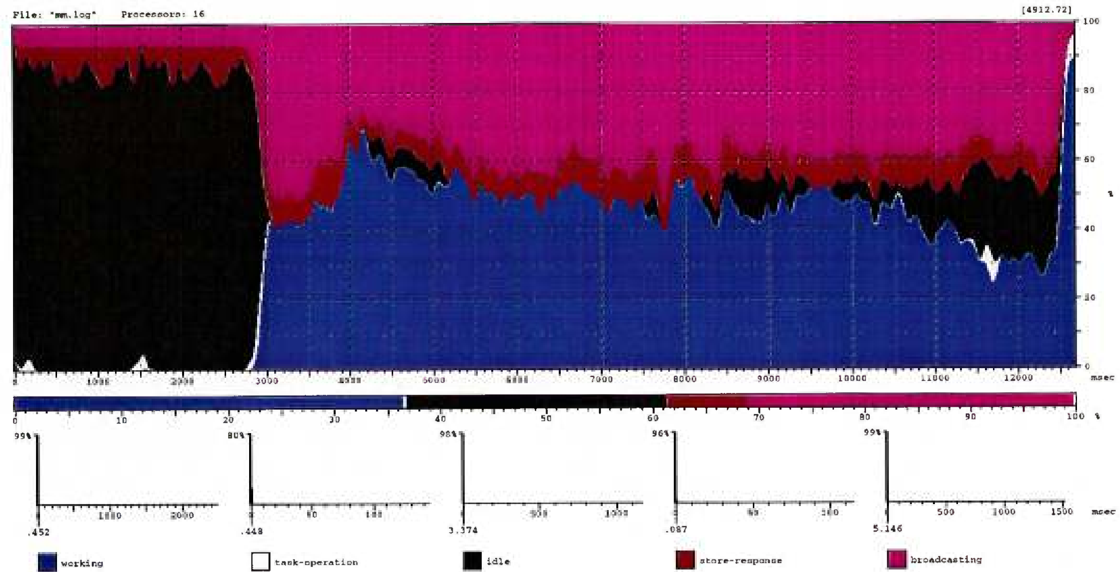

FIG. B.5 - *Profil d'exécution de* mm *sur le multi-ordinateur.*

B.6 mm-variante

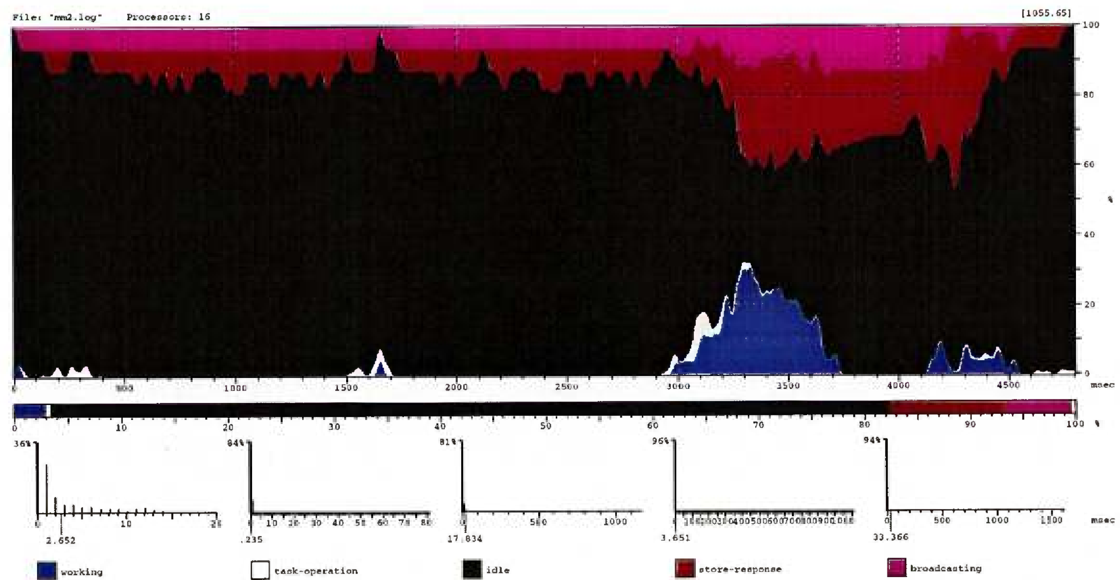

FIG. B.6 - *Profil d'exécution de* mm-variante *sur le multi-ordinateur.*

B.7 abisort

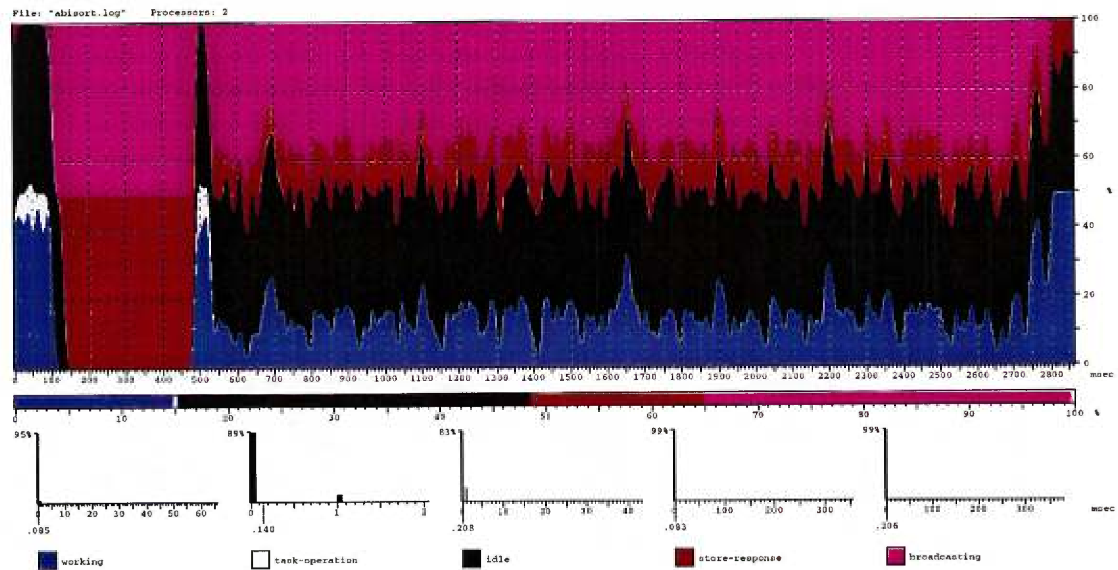

FIG. B.7 - *Profil d'exécution de* abisort *sur le multi-ordinateur.*

B.8 poly

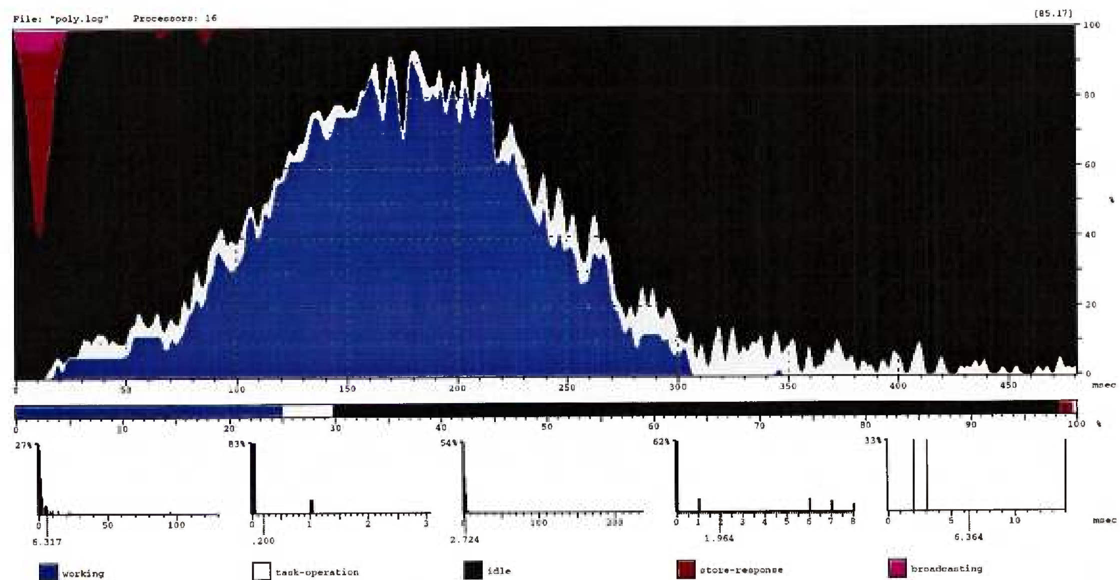

FIG. B.8 - *Profil d'exécution de* poly *sur le multi-ordinateur.*

B.9 scan

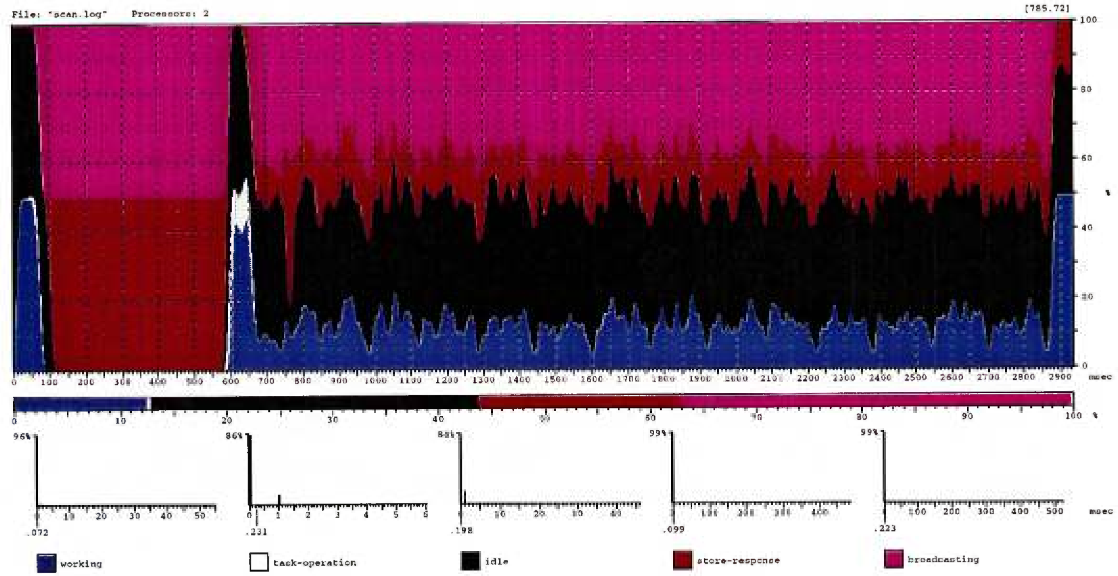

FIG. B.9 - *Profil d'exécution de* scan *sur le multi-ordinateur.*

B.10 sum

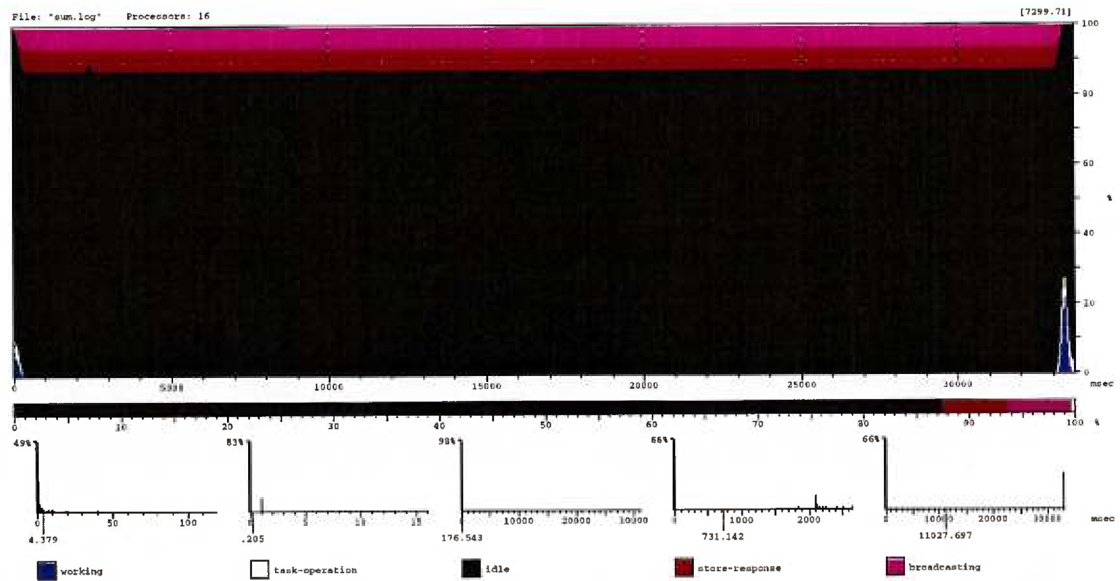

FIG. B.10 - *Profil d'exécution de* sum *sur le multi-ordinateur.*

B.11 Bonne exécution de scan_mt

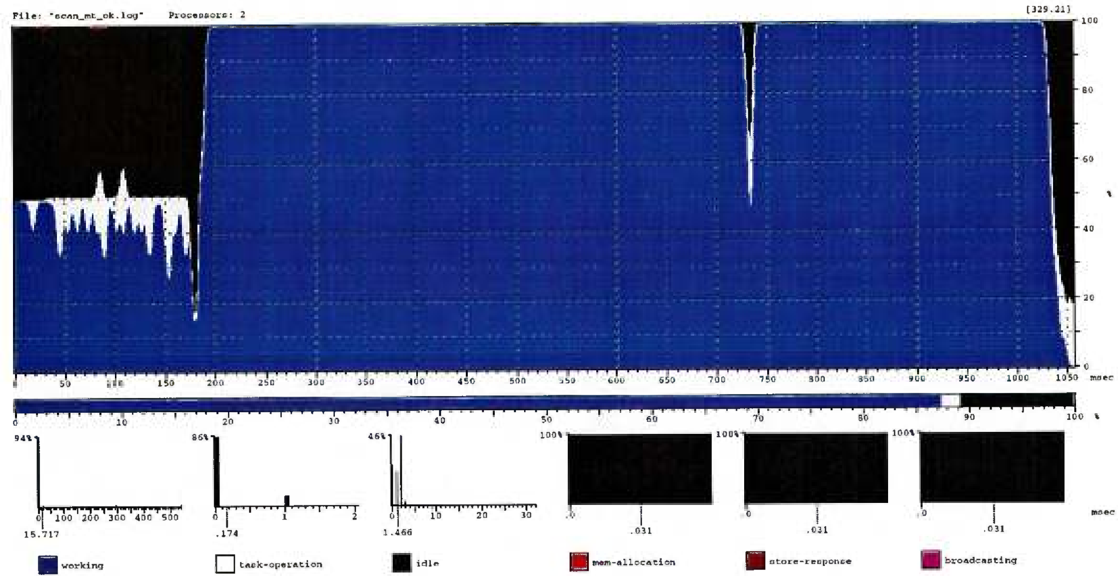

FIG. B.11 - *Profil d'exécution de* scanmtok *sur le multi-ordinateur.*

B.12 Mauvaise exécution de scan_mt

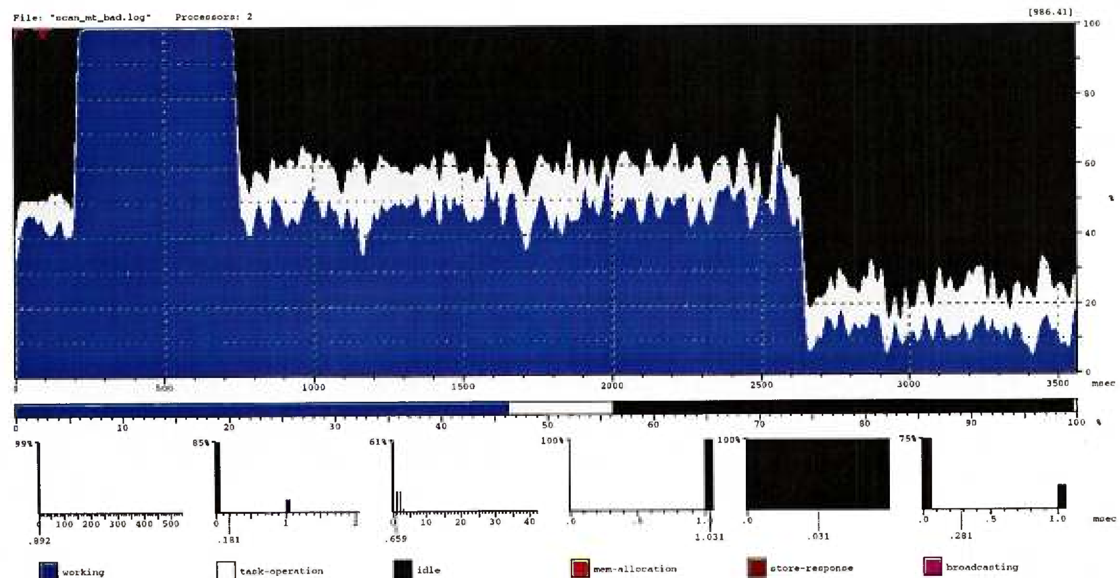

FIG. B.12 - *Profil d'exécution de* scanmtbad *sur le multi-ordinateur.*

Bibliographie

[AGG+] Joshua S. Auerbach, Arthur P. Goldberg, Germán S. Goldszmidt, et al. *Concert/C: A language for distributed programming.* IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

[Amd67] G. Amdahl. Validity of the single processor approach to achieving large-scale computer capabilities. In *AFIPS Conference Proceedings*, volume 30, 1967.

[ARH95] J. H. Reppy A. Rogers, M. C. Carlisle and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, pages 17(2):233–263, March 1995.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, March 1988.

[Bau92] Barr E. Bauer. *Practical Parallel Programming.* Academic Press, 1992.

[BN89] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal of Computing*, April 1989.

[CDGs+] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, et al. *Parallel programming in Split-C.* University of California, Computer Science Division, Berkeley.

[DBF94] V. Van Dongen, C. Bonello, and C. Freehill. High Performance C language specification version 0.8.9. Technical report, Centre de Recherche Informatique de Montréal, April 1994.

[Dig93] Digital Equipment Corporation. *Network Programmer's Guide*, March 1993.

[Fee93] Marc Feeley. *An efficient and general implementation of futures on large scale shared-memory multiprocessors.* PhD thesis, Brandeis University, 1993.

[GM88] Narain Gehani and Andrew D. McGettrick. *Concurrent programming.* International computer science series. Addison-Wesley, 1988.

[GR89] Narain Gehani and William D. Roome. *The Concurrent C programming langage.* Silicon Press, 1989.

[Hal85] R. Halstead. Multilisp: A language for concurrent symbolic computation. In *ACM Trans. on Prog. Languages and Systems*, October 1985.

[HDG94] Gilles Hurteau, Vincent Van Dongen, and Guang Gao. EPPP - an integrated environment for portable parallel programming. Technical report, Centre de Recherche Informatique de Montréal, July 1994.

[Hew92] Hewlett Packard. *Berkeley IPC Programmer's Guide*, 1992.

[HQ91] Philip J. Hatcher and Micheal J. Quinn. *Data-parallel programming on MIMD computers.* Scientific and engineering computation series. The MIT Press, 1991.

[Hwa93] Kai Hwang. *Advanced computer architecture: parallelism, scalability, programmability.* McGraw-Hill, 1993.

[LPT92] O. Lempel, S. Pinter, and E. Turiel. Parallelizing a C dialect for distributed memory MIMD machine. In *Language and Compilers for Parallel Computing*, page 369, August 1992.

[MKH90] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. Technical report, Digital Equipment Corporation, Cambridge Research Lab, November 1990.

[Nik94] R. S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *Languages and Compilers for Parallel Computing*, pages 376–390, August 1994.

[Num94] R. W. Numrich. *The Cray T3D Address Space and How to Use It.* Cray Research Inc., 1994.

[Pol88] Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[RS87] J. Rose and G. Steele Jr. C*: An extended C langage for data parallel programming. In *Proceedings of the Second International Conference on Supercomputing, Vol.2*, May 1987.

[Set90] Ravi Sethi. *Programming Languages: Concepts and Constructs.* Addison Wesley, June 1990.

[SPG92] A. Siberschatz, J. Peterson, and P. Galvin. *Operating System Concepts.* Addison Wesley, third edition, April 1992.

[Wil95] Gregory V. Wilson. *Practical Parallel Programming.* Scientific and Engineering Computation Series. The MIT Press, 1995.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel Vector Computers.* ACM Press Frontier Series. Addison-Wesley Publishing Company, 1991.