2m11.2861.1

Université de Montréal

CPar : Implantation et évaluation d'un compilateur pour une variante parallèle de C.

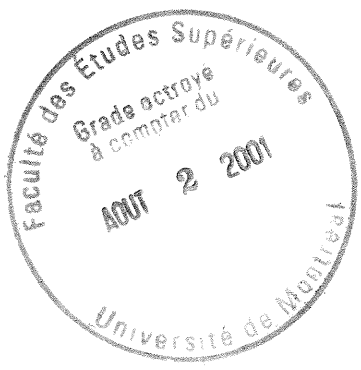
par

Eric Methot

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 science (M.Sc.)
en informatique



Mai 2001

©Eric Methot

QA
76
N54
2001
v.016

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé :

CPar : Implantation et évaluation d'un compilateur
pour une variante parallèle de C.

présenté par :

Eric Methot

a été évalué par un jury composé des personnes suivante :

_____
Jean Vaucher - Président du jury

_____
Marc Feeley - Directeur de recherche

_____
Bernard Gendron - Co-directeur de recherche

_____
Claude Frasson - Membre du jury

Mémoire accepté le : 27 juin 2001

Sommaire

CPar : implantation et évaluation d'un compilateur pour une variante parallèle de C

par Eric Methot

L'intérêt pour la programmation parallèle est croissant. En plus de permettre une utilisation plus efficace des ressources d'un ordinateur, la programmation parallèle offre aussi la possibilité d'accélérer considérablement la vitesse des programmes lorsqu'ils sont exécutés sur des ordinateurs multiprocesseurs. La programmation parallèle reste néanmoins une tâche relativement complexe.

Ce mémoire porte sur la conception et sur l'évaluation d'un compilateur CPar, une extension du langage C qui facilite la création de programmes parallèles. Le langage CPar fournit au programmeur un moyen simple d'exprimer le parallélisme dans ses programmes et ce, surtout lorsqu'ils possèdent une structure récursive. En général, de tels programmes peuvent être partitionnés en tâches (une tâche est simplement une séquence d'instructions machines), à exécuter indépendamment sur les multiples processeurs d'un ordinateur parallèle. Toutefois, la création de ces tâches, leur affectation aux processeurs, leur exécution, leur synchronisation, et leur destruction, engendrent ce que nous appelons un surcoût de gestion, qui peut parfois nuire aux performances des programmes parallèles. L'un des objectifs du compilateur que nous proposons consiste à minimiser l'impact négatif de ce surcoût de gestion.

Le compilateur CPar génère du code C dans le dialecte de *gcc* et ne modifie le code qu'aux endroits où les constructions d'appels parallèles existent. Les étapes importantes dans l'optimisation de la génération de code sont les suivantes :

- Employer une représentation efficace de la pile de tâches.
- Trouver une technique efficace pour la création et l'initialisation des descripteurs de tâches.
- Déterminer la stratégie d'accès aux données locales d'un processeur.
- Implanter efficacement l'empilement et le dépilement d'une tâche.

- Effectuer un balancement automatique de la charge de travail en utilisant la technique du vol de tâches.
- Mettre au point une interface simple pour l'exécution d'une tâche via les fonctions "proxy".
- Implanter un mécanisme simple mais efficace de synchronisation entre les tâches.
- Modifier le protocole d'exclusion mutuelle rapide afin d'éviter l'utilisation d'écriture en mémoire de type "write-through".

Ce sont les tests empiriques qui déterminent l'efficacité de l'implantation, et par conséquent si notre objectif est atteint ou pas. La série de tests effectués démontrent que nous obtenons un surcoût de gestion raisonnablement faible sur la plateforme SPARC, mais que sur Intel les performances sont encore meilleures. Le surcoût de gestion peut être réduit considérablement lorsque l'on prend avantage de l'architecture sous-jacente. CPar permet donc au programmeur d'exploiter le parallélisme dans ses programmes à un coût relativement faible. Le problème de la granularité s'en trouve aussi diminué puisque CPar supporte très bien la création d'un grand nombre de tâches d'une granularité fine. Il peut ainsi lui offrir un balancement automatique et équitable de la charge de travail.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

L'intérêt pour la programmation parallèle est croissant. En plus de permettre une utilisation plus efficace des ressources d'un ordinateur, la programmation parallèle offre aussi la possibilité d'accélérer considérablement la vitesse des programmes lorsqu'ils sont exécutés sur des ordinateurs multiprocesseurs. La programmation parallèle reste néanmoins une tâche relativement complexe.

Ce mémoire porte sur la conception et sur l'évaluation d'un compilateur CPar, une extension du langage C qui facilite la création de programmes parallèles. Le langage CPar fournit au programmeur un moyen simple d'exprimer le parallélisme dans ses programmes et ce, surtout lorsqu'ils possèdent une structure récursive. En général, de tels programmes peuvent être partitionnés en tâches (une tâche est simplement une séquence d'instructions machines), à exécuter indépendamment sur les multiples processeurs d'un ordinateur parallèle. Toutefois, la création de ces tâches, leur affectation aux processeurs, leur exécution, leur synchronisation, et leur destruction, engendrent ce que nous appelons un surcoût de gestion, qui peut parfois nuire aux performances des programmes parallèles. L'un des objectifs du compilateur que nous proposons consiste à minimiser l'impact négatif de ce surcoût de gestion.

Dans ce mémoire nous employons plusieurs termes liés à la programmation parallèle, qu'il est nécessaire de définir ou de clarifier :

noyau: Le noyau de CPar est une bibliothèque qui contient les primitives employées par tous les programmes CPar compilés.

processus lourd: Un processus lourd est un processus au sens traditionnel du terme. Il s'agit d'un programme en exécution avec son propre espace mémoire et un seul flot de contrôle. Ne pouvant partager sa mémoire avec un autre processus lourd, la communication entre deux processus lourds se fait habituellement par l'entremise de fichiers ou de sockets [1].

processus léger: Il est parfois désirable de partager les ressources d'un processus lourd de telle sorte qu'un ensemble de flots de contrôle puissent y accéder concurremment. C'est ainsi que plusieurs systèmes d'exploitation ont introduit la notion de processus léger ("thread" en anglais). Ce dernier consiste en un flot de contrôle avec une unité de temps UCT (unité centrale de traitement), un ensemble de registres et une pile d'exécution. Il partage avec les autres processus légers l'ensemble de l'espace mémoire du processus lourd [1, 17]. Ainsi, la communication entre deux processus légers se fait habituellement par l'entremise de cette mémoire partagée. Notons qu'avec l'introduction des processus légers, un processus lourd peut se définir comme l'union d'un espace mémoire et d'un ensemble de processus légers.

tâche: Pour nos besoins, une tâche est simplement une abstraction d'une séquence d'instructions machine. Ainsi, lorsqu'on dit que deux tâches peuvent être exécutées en parallèle, ce sont les deux séquences d'instructions qui peuvent l'être.

partitionnement: Partitionner un programme consiste à le découper en plusieurs tâches. Le partitionnement peut être statique ou dynamique. Dans le premier cas, le nombre de tâches sera fixe, alors que dans le second, il sera variable et déterminé par un ensemble de facteurs tels que les paramètres d'entrées du programme et le nombre de processeurs disponibles.

processeur logique: Dans le contexte de l'exécution d'un programme parallèle, un processeur logique (à distinguer d'un processeur physique), n'est rien de plus qu'un processus léger qui exécute une tâche. Dans le contexte particulier du compilateur CPar, il s'agit plutôt d'un processus léger qui exécute le code

de notre noyau. Ainsi, dans notre implantation de CPar, ces processeurs logiques sont des automates qui exécutent les tâches d'un programme CPar et qui peuvent communiquer entre eux pour se synchroniser, par exemple.

surcoût de gestion: Pendant l'exécution d'un programme parallèle, il est nécessaire de gérer le cycle de vie, c'est-à-dire la création, la distribution, l'exécution, la synchronisation et la destruction de toutes les tâches. Cette gestion engendre un coût à l'exécution qu'il faut minimiser afin d'obtenir les meilleures performances possibles. L'ensemble des instructions machine qui doivent être exécutées pour effectuer cette gestion font partie de ce que nous nommons le surcoût de gestion.

granularité d'une tâche: La granularité d'une tâche peut se définir comme étant une mesure de sa taille par rapport à la taille du programme en entier (somme de la taille de toutes ses tâches). La taille peut être exprimée en nombre d'instructions, mais aussi en unités de temps ou toute autre mesure appropriée au contexte. Ainsi, lorsque le rapport est petit, on dit que la granularité de la tâche est fine. Lorsqu'il est grand, on dit plutôt que la granularité est grande. Par exemple, une tâche dont la taille est d'une unité de temps est très fine si le programme en compte un milliard. Elle est cependant d'une granularité plus importante dans le contexte d'un programme dont la taille est de dix unités de temps.

balancement de la charge: Lorsqu'un programme est partitionné en plusieurs tâches, il est souhaitable de répartir la charge de travail équitablement parmi tous les processeurs logiques afin d'exploiter le parallélisme au maximum. Faisant partie des activités de gestion du parallélisme, le balancement de la charge est une activité qui consiste à distribuer les tâches sur l'ensemble des processeurs dans le but de maximiser le parallélisme à l'exécution. Il y a plusieurs façons d'effectuer un balancement de la charge. La responsabilité peut reposer sur le programmeur, ce qui rend son travail d'autant plus complexe. Le balancement de la charge peut cependant être partiellement ou complètement automatisé ce qui, au contraire, simplifie son travail.

extensibilité: Dans le contexte de ce mémoire, l'extensibilité (de l'anglais "scalabili-

ty") signifie la capacité à produire un gain de performances proportionnel aux ressources de calcul. Par exemple, on s'attend à ce qu'un programme extensible puisse diviser son temps d'exécution par deux si l'on double le nombre de processeurs (ce pour les mêmes paramêtres d'entrées). Les programmes sont rarement parfaitement extensibles puisque la communication entre processeurs reste un facteur limitatif de l'extensibilité.

Le vocabulaire ayant été précisé, il est maintenant possible d'approfondir notre étude du compilateur et du noyau de CPar.

1.1 Le problème de la granularité des tâches

Dans un contexte où la taille d'un programme est constante le nombre de processeurs est connu et le programme possède une structure régulière (c'est-à-dire essentiellement constitué de boucles imbriquées), il est relativement simple d'effectuer un partitionnement statique du programme de telle sorte que le balancement de la charge soit optimal et le surcoût de gestion réduit au minimum. Toutefois, dans le cas général, l'exécution et le flot de contrôle dépendent étroitement des données du programme, et ni la taille du programme, ni le nombre de processeurs disponibles ne sont connus. Par conséquent, pour maximiser le parallélisme, il est préférable d'effectuer un partitionnement dynamique du programme.

Il n'est cependant pas simple de partitionner un programme dynamiquement car la granularité des tâches engendrées peut avoir un impact important sur les performances du programme. Un grand nombre de tâches d'une granularité fine peut générer un surcoût de gestion élevé, puisque le temps passé à effectuer la gestion des tâches peut devenir important en comparaison du temps d'exécution du programme. À l'opposé, un nombre insuffisant de tâches dont la granularité est telle qu'une distribution uniforme de la charge s'avère impossible empêche d'exploiter le parallélisme à son maximum. Dans tous les cas, l'impact est négatif et les performances en sont réduites.

La figure 1.1 illustre ces deux extrêmes. Il s'agit d'un même programme dont la

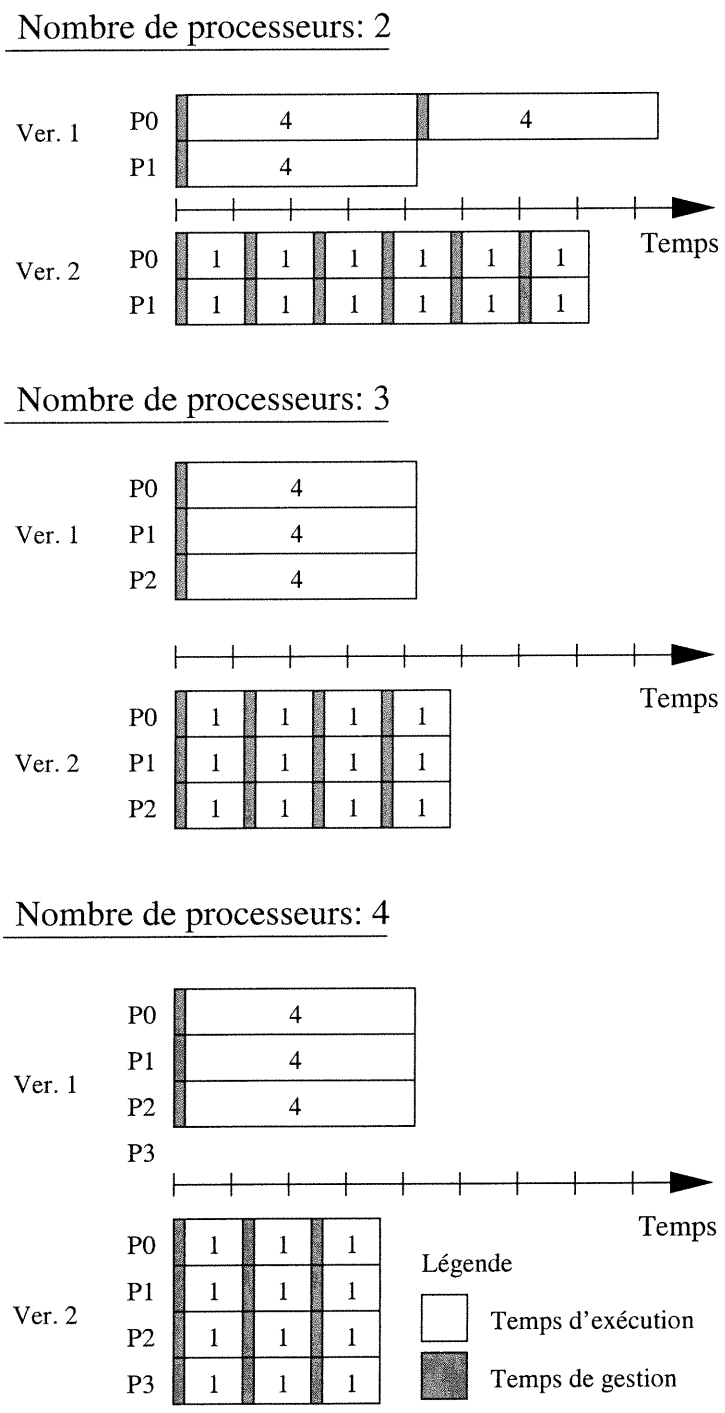


FIG. 1.1 – L'impact de la granularité sur l'exécution d'un programme

première version (Ver. 1) est partitionnée en trois tâches de taille 4 et la seconde version (Ver. 2) en 12 tâches de taille 1. Les deux versions du programmes sont exécutées sur des ordinateurs ayant respectivement deux, trois et quatres processeurs. Comme on peut le constater, aucune des versions n'est systématiquement meilleure que l'autre. La première version, dont les tâches sont d'une granularité plus grande, engendre un surcoût de gestion moins important mais s'exécute parfois moins rapidement à cause d'une utilisation moins efficace des processeurs. La seconde version, dont les tâches sont d'une granularité plus fine engendre un surcoût de gestion plus important mais utilise pleinement tous les processeurs à sa disposition. Toutefois, le surcoût de gestion peut parfois réduire ses performances.

Remarquons que si le balancement de la charge n'est pas toujours optimal, c'est que dans certains cas la granularité des tâches est trop grande ou leur nombre est insuffisant. Il est évident qu'un grand nombre de tâches fines offre une liberté dans la distribution des tâches qui facilite le balancement de la charge. Cependant, cette liberté vient au prix d'un surcoût de gestion plus élevé. En réduisant substantiellement le surcoût de gestion, on pourrait créer un grand nombre de tâches d'une granularité relativement fine. On peut ensuite effectuer un balancement de la charge efficace pour s'approcher d'une performance optimale. Afin de tirer pleinement avantage de cette stratégie, il est toutefois nécessaire de pouvoir automatiser le balancement de la charge. Sinon, le travail incomberait au programmeur et il peut devenir complexe lorsque le nombre de tâches est important.

1.2 Le contexte de développement de CPar

Le nom "CPar" est une contraction de "C" et de "Parallèle", soulignant ainsi que le langage CPar est une variante parallèle du langage C. Le langage permet de décrire le parallélisme de contrôle de manière simple et concise. La principale extension syntaxique au langage C est le mot réservé `par` qui permet d'effectuer un appel de fonction parallèle. La figure 1.2 illustre comment un appel parallèle est exprimé en CPar. Il suffit simplement d'ajouter après un appel de fonction le mot réservé `par` et un bloc d'énoncés pour exprimer l'exécution parallèle de l'appel et du bloc

```
fonction(arg1,...,argc) par { ... }
```

FIG. 1.2 – Syntaxe d'un appel parallèle CPar

```
{
  A;
  f() par { B; };
  C;
}
```

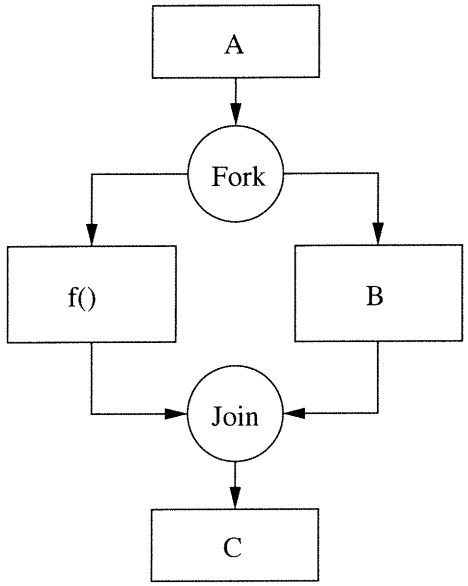


FIG. 1.3 – La sémantique d'un appel parallèle de CPar est de type "fork/join"

d'énoncés.

L'expression que l'on retrouve à la figure 1.2 signifie donc que l'appel de fonction (une tâche) peut être exécuté en parallèle avec le bloc d'énoncés (une autre tâche). Le résultat de cette expression est celui retourné par la fonction et de plus, il y a une synchronisation implicite à la fin de l'expression. Une expression parallèle dans CPar suit donc une sémantique de type "fork/join" comme illustrée à la figure 1.3. Ainsi, l'appel de fonction et le bloc d'énoncés sont séparés en deux flots de contrôle distincts. Ceux-ci seront fusionnés à nouveau avant de procéder à l'exécution du reste du programme.

On constate aussi que la syntaxe d'une expression parallèle de CPar permet au langage de traiter toutes les tailles de tâches. Même s'il y a une limite inférieure à la taille d'un appel de fonction, il n'y en a aucune pour un bloc d'énoncés. De plus, il n'y a clairement aucune limite supérieure sur la taille de ces tâches. Par conséquent, il est effectivement possible de traiter toutes les tailles de tâches.

```
1  int fib(int n) {
2    if ( n < 2)
3      return 1;
4    else {
5      int f1, f2;
6      f1 = fib(n-1) par {
7        f2 = fib(n-2);
8      };
9      return f1 + f2;
10   }
11 }
```

FIG. 1.4 – Exemple d'un programme CPar

```
1  #ifndef __CPAR__
2  #define par;
3  #endif
```

FIG. 1.5 – Transformation d'un programme CPar en un programme C

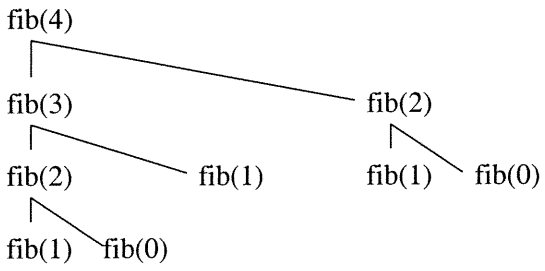


FIG. 1.6 – L'arbre de tâches engendré par un appel à fib(4)

La figure 1.4 donne un exemple d'un programme CPar simple qui calcule un nombre de la suite de Fibonacci. Notons que ce programme emploie un algorithme naïf mais permet toutefois d'illustrer certains principes du langage avec facilité.

D'abord, remarquons que la fonction `fib` exprime le calcul parallèle des deux appels récursifs `fib(n-1)` et `fib(n-2)` et que le seul élément qui distingue ce programme d'un programme C est le mot réservé `par` qu'on y retrouve à la ligne 6. En effet, en remplaçant celui-ci par un `;` , on obtient un programme C séquentiel équivalent (qui fait le même calcul). Cette substitution peut normalement avoir lieu sans crainte afin d'obtenir un programme séquentiel équivalent. Il existe néanmoins deux exceptions. La première, le programme ne sera pas équivalent si l'ordre d'évaluation des tâches est important. La seconde, si la substitution des symboles engendre une erreur de syntaxe. Par exemple, l'expression parallèle `if (f(&x) par {x++;}) y;` possède ces deux caractéristiques. D'une part, le résultat dépend de l'ordre d'évaluation des deux tâches (on suppose que `f` utilise x). De l'autre, en remplaçant le symbole `par` par le symbole `;` un compilateur C conventionnel génèrerait une erreur de syntaxe. Hormis ces deux conditions, il est facile de compiler un programme CPar avec un compilateur C conventionnel. À cette fin, le préprocesseur C nous est très utile et la figure 1.5 donne les quelques lignes de code qui lui permettent d'effectuer cette substitution de symboles automatiquement.

La fonction de Fibonacci à la figure 1.2 utilise une approche "diviser pour régner". Grâce à la récursivité, nous pouvons prendre pleinement avantage de la puissance de CPar. Ainsi, l'appel parallèle n'engendre pas que deux tâches, mais bien un arbre de tâches, chacune étant associée à un appel de fonction. La figure 1.6 illustre l'arbre de tâches engendré par un appel à fib(4). Comme on peut le constater, il peut y avoir jusqu'à cinq appels en parallèle si les feuilles sont exécutées simultanément. Cependant, avec fib(9) ce nombre serait de 55 et pour fib(17) il serait de 2584.

La syntaxe illustrée à la figure 1.2 est la même que celle de ParSubC [13], l'ancêtre de CPar, sauf que le symbole `par` est remplacé par le symbole `! !`. Par conséquent, il hérite des caractéristiques de ParSubC qui sont : simplicité, souplesse et puissance. Le langage CPar est une extension modeste de C au point de vue syntaxique. Il est donc

facile à apprendre et à utiliser. Comme nous avons pu le constater lors de l'exemple précédent, la récursivité permet un partitionnement dynamique du programme [13]. Ainsi, le nombre et la taille des tâches peuvent varier avec la taille du problème. Le langage CPar est donc particulièrement bien adapté à un style de programmation "diviser pour régner". Comme la classe des programmes pouvant être traité ainsi est vaste, cela confère à CPar une certaine souplesse. Lorsque l'on combine simplicité et souplesse au faible surcoût de gestion et au balancement automatique de la charge que procurent notre compilateur CPar et son noyau, on obtient un système de programmation efficace qui caractérise la puissance de notre implantation du langage CPar. Noton cependant que, contrairement au compilateur ParSubC [13], notre compilateur CPar ne permet pas de distribuer le calcul sur un réseau d'ordinateurs, car il est conçu spécifiquement pour fonctionner sur les multiprocesseurs à mémoire partagée.

Finalement, on constate que le programme de la figure 1.4 est d'une granularité fine. En effet, la fonction de Fibonacci récursive est l'une des fonctions qui nécessite le moins d'efforts de calculs. Elle agit en quelque sorte comme une borne inférieure à la granularité des fonctions qui possèdent un haut degré de parallélisme (qui engendrent un grand nombre de tâches parallèles) et qui soient calculables efficacement par notre implantation du langage CPar. La fonction constitue donc un test important dans l'évaluation des performances du compilateur et du noyau de CPar.

1.3 Objectifs et contributions

CPar possède au départ des caractéristiques intéressantes qui lui sont conférées par ParSubC, son ancêtre, qui est une extension parallèle d'un sous-ensemble de C. L'objectif de notre travail consiste à construire un compilateur CPar performant. Les difficultés principales, comparativement à ParSubC, consistent à traiter le langage C au complet et à produire un code cible optimisé. Nous nous fixons également un autre objectif de seconde importance qui est de maintenir une portabilité élevée. Pour atteindre ces objectifs il faut :

1. Minimiser le coût de la création et de la gestion d'une tâche.
2. Minimiser le coût de la synchronisation implicite entre les tâches.
3. Effectuer un balancement automatique de la charge de façon efficace.
4. Centraliser dans un fichier tous les aspects non portables du noyau de CPar.

Le principal attrait de notre compilateur CPar réside donc au niveau des performances des programmes générés par son compilateur. Comme on pourra le constater, plusieurs stratégies sont employées afin de minimiser l'impact négatif du surcoût de gestion. Combinée à une technique éprouvée pour le balancement automatique de la charge, elles permettent d'obtenir des performances intéressantes.

1.4 Aperçu

Avant d'exposer le fonctionnement interne du compilateur CPar, il convient de situer le langage CPar, ainsi que son implantation, à l'intérieur de la grande famille des langages parallèles. Ainsi, le prochain chapitre de ce mémoire propose une revue de la littérature sur les langages parallèles. Ensuite, dans le chapitre 3, nous sommes en mesure de présenter la réalisation du compilateur et du noyau, puis dans le chapitre 4, l'évaluation de ses performances. Finalement, le dernier chapitre résume les aspects importants de l'implantation du compilateur CPar et propose l'exploration de certaines avenues de recherche dans le but d'améliorer le compilateur ainsi que ses performances.

Le chapitre 2 est une revue de la littérature qui présente un survol des langages et bibliothèques pour la programmation parallèle, ainsi qu'un aperçu des langages parallèles spécialisés semblables à CPar. On motive l'utilité de ce nouveau langage et on y constate que CPar permet de combler certaines lacunes des autres langages, sans toutefois posséder l'ensemble de leurs qualités.

Le chapitre 3 décrit les différentes techniques employées par le compilateur CPar et son noyau afin de minimiser le surcoût de gestion. On y présente le modèle

général d'exécution, ainsi que les structures de données, la méthode d'accès aux données locales des processus légers, la création paresseuse des tâches et enfin, le vol des tâches. Certaines optimisations moins portables sont aussi présentées.

Le chapitre 4 présente une évaluation empirique des programmes générés par le compilateur CPar. On y expose entre autres les résultats des tests sur un grand nombre de programmes de calculs numériques et de recherche opérationnelle exécutés sur deux plateformes différentes (SPARC et Intel). Les performances de CPar sont aussi comparées à celles de Cilk [16] et de ParSubC.

Le chapitre 5 conclut en récapitulant les points importants. L'exploration de certaines voies est aussi proposée en vue d'améliorer le langage et les performances de son compilateur.

Chapitre 2

Revue de littérature

L'écriture de programmes parallèles reste une activité complexe. Cette complexité croît d'autant plus si l'on s'attend à ce que les performances de ces programmes augmentent proportionnellement au nombre de processeurs et ce, de manière automatique. Il existe plusieurs langages de programmation et des bibliothèques qui facilitent la programmation parallèle. Nous proposons ici un survol de certains d'entre eux.

Ainsi, on peut classifier ces langages (ou bibliothèques) en deux catégories : la première contient les langages généraux et la seconde, ceux qui se spécialisent dans la programmation parallèle. Dans les deux cas, il existe un bon nombre de candidats représentatifs pour chacune de ces catégories.

Par définition, un langage général ne possède pas les mêmes objectifs qu'un langage spécialisé. Il est donc difficile de les comparer avec un même ensemble de critères. Il nous semble important de pouvoir exprimer facilement le parallélisme avec un coût à l'exécution qui soit relativement peu élevé. Nous allons donc nous concentrer sur la simplicité des langages et la performance de leur implantation.

Nous proposons donc un survol de plusieurs langages et bibliothèques. Nous les examinerons à des degrés de détails variables afin d'en déterminer leurs forces et leurs faiblesses. Cet examen nous permettra de mieux situer CPar par rapport aux autres langages parallèles.

2.1 Bibliothèques et langages généraux

Il existe deux bibliothèques bien connues qui facilitent l'écriture de programmes parallèles et distribués en C ou en C++ : MPI ("Message Passing Interface") [6] et PVM ("Parallel Virtual Machine") [2]. Ces deux bibliothèques s'appuient sur l'échange de messages pour leurs communications et sont efficaces lorsque la granularité des tâches est grande. Pour des problèmes qui nécessitent une granularité plus fine, le langage Java et la biliothèque *pthreads*, une implantation du standard POSIX 1003.1c-1995, sont plus intéressants car ils sont basés sur les processus légers et une communication par l'entremise d'une mémoire partagée.

2.1.1 Message Passing Interface (MPI)

Un consortium de fabricants et de scientifiques ont mis au point la bibliothèque MPI, un standard pour l'écriture de programmes parallèles utilisant l'échange de messages. Le but de cette bibliothèque consiste à faciliter la création de programmes portables et efficaces en employant l'échange de messages comme unique moyen de communications. Elle s'adresse plutôt à la programmation distribuée ou à la programmation parallèle sur une architecture à mémoire distribuée, sans pour autant s'y limiter. La bibliothèque est d'ailleurs en constante évolution. Une série d'extensions ont été ajoutées à la bibliothèque originale pour former le standard MPI-2 [6, 7] (voir http ://www.mcs.anl.gov/mpi/).

Un programme MPI regroupe habituellement les tâches par sous-ensembles et une tâche peut appartenir à un nombre arbitraire d'ensembles. La bibliothèque MPI offre donc plusieurs fonctions permettant la création et la manipulation de ces ensembles de tâches. Ces fonctions comprennent l'inclusion, l'exclusion, l'union, l'intersection, le complément, la soustraction et d'autres encore. L'un des prérequis d'un système qui emploie l'échange de messages est de garantir une communication fiable avec laquelle il est possible de distinguer les messages d'intérêt des autres. À ce titre, la bibliothèque MPI introduit la notion de "communicateur". Ce dernier est responsable des communications à l'intérieur d'un groupe de tâches.

Le surcoût de gestion d'une tâche MPI est élevé. D'abord parce que la création d'une tâche MPI requiert la création d'un processus lourd et ensuite parce que la synchronisation, qui se fait par échange de messages, est coûteuse. Pour éviter que le surcoût de gestion ne prenne une trop grande proportion du temps d'exécution, le nombre de tâches MPI sur un même ordinateur doit être proche du nombre de processeurs physiques et leur granularité suffisamment importante.

La bibliothèque MPI reste donc bien adaptée à un environnement à mémoire distribuée et s'adresse surtout à la classe de programmes dont les tâches sont d'une granularité relativement grande.

2.1.2 Parallel Virtual Machine (PVM)

Le système PVM [2] permet de configurer un ordinateur parallèle à mémoire partagée en employant un réseau d'ordinateurs possiblement hétérogènes. L'une des forces de PVM consiste à rendre transparent à l'usager l'emplacement physique des tâches. Par conséquent, que deux tâches se trouvent sur un même ordinateur ou sur des ordinateurs distincts, elles peuvent communiquer entre elles de la même manière.

Un autre aspect intéressant de la bibliothèque PVM consiste à permettre une très grande portabilité des programmes. En effet, la bibliothèque offre une grande flexibilité au niveau de l'envoi de messages dont la structure peut être arbitrairement complexe. La bibliothèque permet de partager de l'information d'une tâche à l'autre de façon transparente, même si les représentations physiques des données peuvent varier d'un ordinateur à l'autre. Par exemple, la représentation des entiers sur Intel ("little endian") n'est pas la même que sur SPARC ("big endian"). PVM offre les fonctionnalités qui permettent de faire abstraction de ce genre de détails.

Tout comme la bibliothèque MPI, le surcoût de gestion d'une tâche est assez élevé. Cela signifie que la bibliothèque PVM se prête aussi à des problèmes dont la granularité des tâches est relativement grande. PVM s'avère donc un excellent outil pour le calcul parallèle et distribué à grande échelle et dont la granularité des tâches permet la distribution du calcul comme, par exemple, la factorisation de grands nombres.

2.1.3 Java

Le langage Java est un langage orienté objet portable développé par Sun Microsystems en 1991. L'exécution des programmes Java se fait par l'intermédiaire d'une machine virtuelle nommé JVM ("Java Virtual Machine"). Ainsi, un compilateur Java traduit le code source Java en un code spécialisé que la JVM pourra interpréter. Certaines implantations peuvent cependant traduire ce code spécialisé en code natif afin qu'il soit plus performant à l'exécution.

La programmation parallèle au sens large est relativement simple en Java car la notion de processus léger est encapsulée dans une classe de base du langage. La gestion et la synchronisation des tâches sont donc intégrées au langage. Pour créer un processus léger, il suffit de créer un objet d'une classe dérivée de `java.lang.Thread` et d'appeler sa méthode `start()`. Pour la synchroniser avec une autre tâche, nous n'avons qu'à appeler la méthode `join()` de cette dernière. Nous possédons tous les outils nécessaires pour implanter le partitionnement dynamique d'un programme.

Le programme à la figure 2.1 illustre une méthode de partitionnement dynamique dans le calcul d'un nombre de la suite de Fibonacci. Ce programme emploie une sémantique "fork/join" identique à celle de CPar et, comme on peut le constater, elle est relativement simple à implanter. Nous définissons d'abord une classe dérivée de `java.lang.Thread` (ligne 3) puis nous fournissons une implantation de la méthode `start()`. Comme pour l'exemple en CPar, cette dernière appelle récursivement l'exécution parallèle de `fib(n-1)` et `fib(n-2)` en créant toutefois deux objets dérivés de `java.lang.Thread` et en appelant leur méthode `start()` (lignes 15 à 18). Le "join" doit cependant être effectué manuellement (lignes 21 et 22).

Notons d'abord qu'en Java la création d'un objet s'effectue toujours sur le tas. À la création, un certain espace mémoire est alloué puis initialisé à zéro. On appelle ensuite le constructeur de l'objet afin de l'initialiser. Sachant de plus que la mémoire est gérée par un "garbage collector", il est clair que la création d'un objet nécessite un grand nombre d'instructions. En particulier, ceux qui sont dérivés de la classe `java.lang.Thread` ne font pas partie du calcul en soit, mais encapsulent la notion de processus légers. Pour ceux-là, les instructions font donc partie du surcoût de

gestion. De plus, comme la création d'un objet dérivé de `java.lang.Thread` entraîne éventuellement la création d'un processus léger, il y a un danger de surcharger le système d'exploitation. Dans un tel scénario, le temps UCT devient presque entièrement dédié à des tâches administratives. Il s'ensuit que le surcoût de gestion pour une telle application reste considérable et que l'impact sur les performances devient d'autant plus important.

Ainsi, de par la gestion de sa mémoire et du lancement possible d'un trop grand nombre de processus légers, Java s'avère un langage mieux adapté au partitionnement statique qu'au partitionnement dynamique.

2.1.4 Posix Threads

Le standard POSIX 1003.1c-1995, mieux connu sous le nom *Posix Threads*, est un "API" pour la gestion de processus légers. L'implantation la plus populaire sous Linux se nomme *pthreads*. Celle-ci est une bibliothèque pour les langages C et C++ qui fournit un type de données abstraites encapsulant la notion de processus légers. Ainsi, *pthreads* est à C et C++ ce que la classe `java.lang.Thread` est à Java. Cette bibliothèque présente donc sensiblement les mêmes avantages et inconvénients. Même si la gestion de la mémoire est moins lourde en C, la nécessité d'initialiser les structures de données (équivalent des constructeurs) et de créer des processus légers pour chaque tâche entraîne un surcoût de gestion trop élevé pour les applications dont les tâches sont d'une granularité fine.

2.2 Les langages parallèles

Les langages et les bibliothèques mentionnés jusqu'à présent comportent certaines faiblesses lorsque l'on veut développer des programmes extensibles. Notamment, il est nécessaire de faire un usage judicieux des tâches parallèles car leur surcoût de gestion est important et, en grand nombre, celles-ci peuvent surcharger le système d'exploitation. Par conséquent, les tâches doivent être d'une granularité suffisamment grande. De plus, la responsabilité du balancement de la charge repose sur

```
 1  import java.io.*;
 2
 3  public class FibThread extends Thread {
 4
 5    int n = 0;
 6    int result = 1;
 7
 8    public FibThread(int n) { this.n = n;    }
 9    public int value()      { return result; }
10
11    public void start() {
12
13      if ( n > 1 ) {
14
15        FibThread th1 = new FibThread(n-1);
16        th1.start();
17        FibThread th2 = new FibThread(n-2);
18        th1.start();
19
20        try {
21          th1.join();
22          th2.join();
23        }
24         catch(Exception e) {
25           // Ignore exceptions
26         }
27
28        result = th1.result + th2.result;
29      }
30    }
31
32    public static void main(String args[]) {
33      int n = Integer.parseInt(args[0]);
34      FibThread fib = new FibThread(n);
35
36      fib.start();
37      try {
38        fib.join();
39      }
40      catch(Exception e) {
41      }
42
43      System.out.println("Fib("+n+")="+fib.value());
44    }
45  }
```

FIG. 2.1 – Un partitionnement dynamique en Java

le programmeur. Ce problème est d'autant plus complexe lorsque la taille du programme et le nombre de processeurs n'est pas connu à l'avance. C'est ici qu'entrent en jeu les langages parallèles.

Afin de maximiser le parallélisme, il est préférable d'effectuer un partitionnement dynamique du programme. Celui-ci peut cependant engendrer un grand nombre de tâches de granularité fine. Or, pour supporter un aussi grand nombre de tâches, il s'avère essentiel d'utiliser des techniques alternatives. Il est clairement inacceptable de créer un processus léger pour chaque tâche lorsque l'ordinateur ne possède qu'un nombre relativement faible de processeurs. D'une part, l'effort de création de ces tâches risque d'être fortement disproportionné par rapport à l'effort de calcul réel. D'autre part, la surcharge du système d'exploitation devient inévitable.

Ces techniques alternatives ont recours à un nombre limité de processus légers qui agissent à titre de processeurs logiques. Leur nombre ne dépasse pas, ou de peu, celui des processeurs physiques afin de limiter le surcoût de gestion. Ainsi, lors de la création d'une tâche, ces processeurs logiques n'effectuent qu'une manipulation de structures de données afin de mémoriser l'avènement d'un "fork" ou d'un "join". Ces processeurs logiques peuvent aussi communiquer entre eux dans le but de se synchroniser ou d'effectuer un balancement de la charge, par exemple.

La "création paresseuse de tâches" [8, 13], ainsi que le "vol de tâches" [9, 13], sont deux techniques qui permettent d'effectuer une gestion efficace des tâches. La première permet de créer un très grand nombre de tâches en employant un nombre limité de processus légers. La seconde permet aux processeurs logiques d'effectuer un balancement efficace et automatique de la charge. Après un bref aperçu de ces techniques, nous portons notre attention sur trois langages parallèles : StackThread/MP, Cilk et ParSubC. Ces trois implantations emploient une variante des techniques de création paresseuse des tâches et de vol de tâches. Elles sont donc en mesure de supporter la création d'un grand nombre de tâches, ainsi qu'un balancement de la charge automatisé. Ces langages seront enfin comparés à CPar afin de déterminer la place qu'occupe ce dernier dans la famille des langages parallèles.

2.3 La création paresseuse de tâches

La création paresseuse de tâches est une technique que l'on peut utiliser pour demander l'exécution d'une tâche sans avoir recours au lancement d'un nouveau processus léger. Dans une approche directe, sans création paresseuse de tâches, la rencontre d'un "fork" dans le flot de contrôle engendre le lancement d'un processus léger. Le lancement d'un processus léger reste cependant une opération coûteuse qui demande, en plus, une gestion de la part du système d'exploitation. En employant la création paresseuse de tâches, un "fork" ne génère plus qu'une structure de données. Cette structure représente une "potentielle" exécution parallèle. C'est au système de décider au moment opportun si la tâche est exécutée séquentiellement à la suite d'une autre tâche, ou en parallèle avec d'autres tâches. Cette approche permet désormais de créer des millions de tâches "parallèles" à un coût raisonnable et ce, sans surcharger le système d'exploitation.

Pour implanter cette technique on emploie habituellement un nombre restreint de processus légers, qui se rapproche du nombre de processeurs physiques. Ceux-ci agissent alors comme des processeurs logiques d'un point de vue conceptuel. Un tel processeur logique exécute une à une les tâches dont il a la responsabilité. Cependant, ces processeurs logiques travaillent tous en parallèle. La figure 2.2 illustre un exemple de ce modèle d'exécution. Nous y voyons cinq processeurs logiques ayant chacun un ensemble de tâches parallèles et qui ne sont supportés que par quatre processeurs physiques.

2.4 Le vol de tâches

Le vol de tâches est une technique qui permet d'obtenir un balancement automatique de la charge. Le vol d'une tâche consiste simplement à transférer une tâche d'un processeur à un autre de telle sorte que son exécution éventuelle devienne la responsabilité d'un autre processeur. C'est en répartissant ainsi les tâches parmi tous les processeurs que l'on obtient le résultat désiré : un balancement équitable de la charge. La figure 2.3 illustre le procédé du vol des tâches.

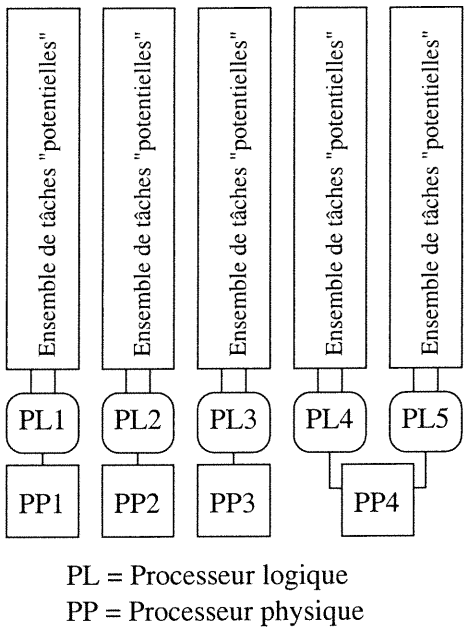


FIG. 2.2 – Un modèle d'exécution qui utilise la création paresseuse de tâches

Certaines implantations emploient un mécanisme d'échantillonnage pour initier le vol d'une tâche. D'autres utilisent plutôt un protocole qui permet aux processeurs d'accéder aux ensembles de tâches des autres processeurs. À ce titre, StackThreads/MP et ParSubC emploient la première approche, tandis que Cilk et CPar utilisent la seconde.

2.5 StackThreads/MP

StackThreads/MP est une extension de C développée par une équipe de chercheurs de l'Université de Tokyo [18]. Le langage possède plusieurs caractéristiques intéressantes, dont l'absence d'un compilateur dédié, et sa technique de manipulation de tâches. De plus, StackThreads/MP ressemble plus à une bibliothèque qu'à un langage de programmation, même s'il n'en est pas tout à fait une. Ce langage implante le vol de tâches, ainsi qu'une technique qui s'apparente à la création paresseuse de tâches.

La figure 2.4 illustre le code de la fonction de Fibonacci récursive. Pour créer une

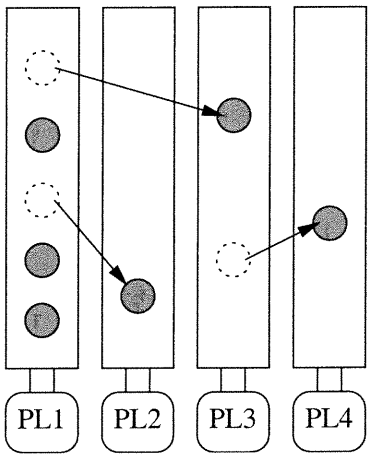


FIG. 2.3 – La répartition de la charge par le biais du vol de tâches

```
1  void pfib(int n, int *r, st_join_counter_t *c) {
2
3    if (n < 2) {
4      *r = 1;
5     st_join_counter_finish(c, 1);
6    }
7    else {
8      int a, b;
9      st_join_counter_t cc[1];
10
11     /* Check if asking for work and free stack  */
12     ST_POLLING();
13
14     /* Initialize join counter */
15     st_join_counter_init(cc, 2);
16
17     /* Fork fib(n-1, ...) */
18     ST_THREAD_CREATE(pfib(n - 1, &a, cc));
19
20     /* Do fib(n-2, ...) */
21     pfib(n-2, &b, cc);
22
23     /* Wait for child's completion */
24     st_join_counter_wait(cc);
25
26     *r = a + b;
27     st_join_counter_finish(c);
28
29     /* Do load balancing & cleanup */
30     ST_POLLING();
31   }
32 }
```

FIG. 2.4 – Exemple d'un programme StackThread

tâche il faut utiliser la construction `ST_THREAD _CREATE(expr)` comme on le voit à la ligne 18. L'expression entre les parenthèses doit être un appel de fonction. Cependant, le langage impose une restriction sur celui-ci : ses arguments ne peuvent pas faire, directement ou indirectement, appel à des primitives du langage. Par exemple, l'expression `ST_THERAD_CERATE(pfib(pfib(2)))` est illégale car `pfib(2)` fait appel à plusieurs primitives. Néanmoins, StackThreads possède d'autres caractéristiques, à notre avis, peu avantageuses.

D'une part, il force le programmeur à effectuer manuellement la synchronisation des tâches. Dans l'exemple de la fonction de Fibonacci de la figure 2.4, c'est le rôle que joue le compteur `cc`. Il est d'abord initialisé à deux, puis chacun des appels récursifs va lui soustraire un à l'aide d'un appel à `st_join_counter_finish(cc,1)`. Le point de synchronisation a lieu au site de l'appel `st_join_counter_wait(cc)`. En plus de rendre la programmation plus complexe, l'utilisation de ce compteur engendre un surcoût non négligeable lorsque la granularité des tâches est fine.

D'autre part, le balancement de la charge n'est pas entièrement automatique car il demande des appels périodiques à la macro `ST_POLLING`. Un des rôles de cette macro, qui s'effectue au coût d'une dizaine d'instructions, consiste à répondre aux demandes de vol de tâches de la part des autres processeurs. Par conséquent, si le programmeur n'effectue pas assez souvent d'appels à cette macro, le balancement de la charge, et donc les performances, en souffrent.

Cependant, la richesse des primitives de synchronisation est un aspect positif de StackThreads. On compte parmi celles-ci les compteurs, les sémaphores, les mutex, les variables de conditions, ainsi que les "spin locks". Le langage offre aussi des primitives qui permettent l'incrémentation atomique. StackThreads rend possible aussi la création de groupes et de hiérarchies de tâches. En plus de ce riche ensemble de fonctionnalités, StackThread/MP ajoute à sa bibliothèque un allocateur dynamique efficace pour remplacer `malloc`. Cet allocateur agit aussi à titre de ramasse miettes ("garbage collector"), ce qui constitue, à notre avis, un avantage significatif.

La compilation des programmes StackThreads est assez inhabituelle. Ce compilateur est en réalité un script qui fait appel à *gcc* pour produire le code assembleur. Ce

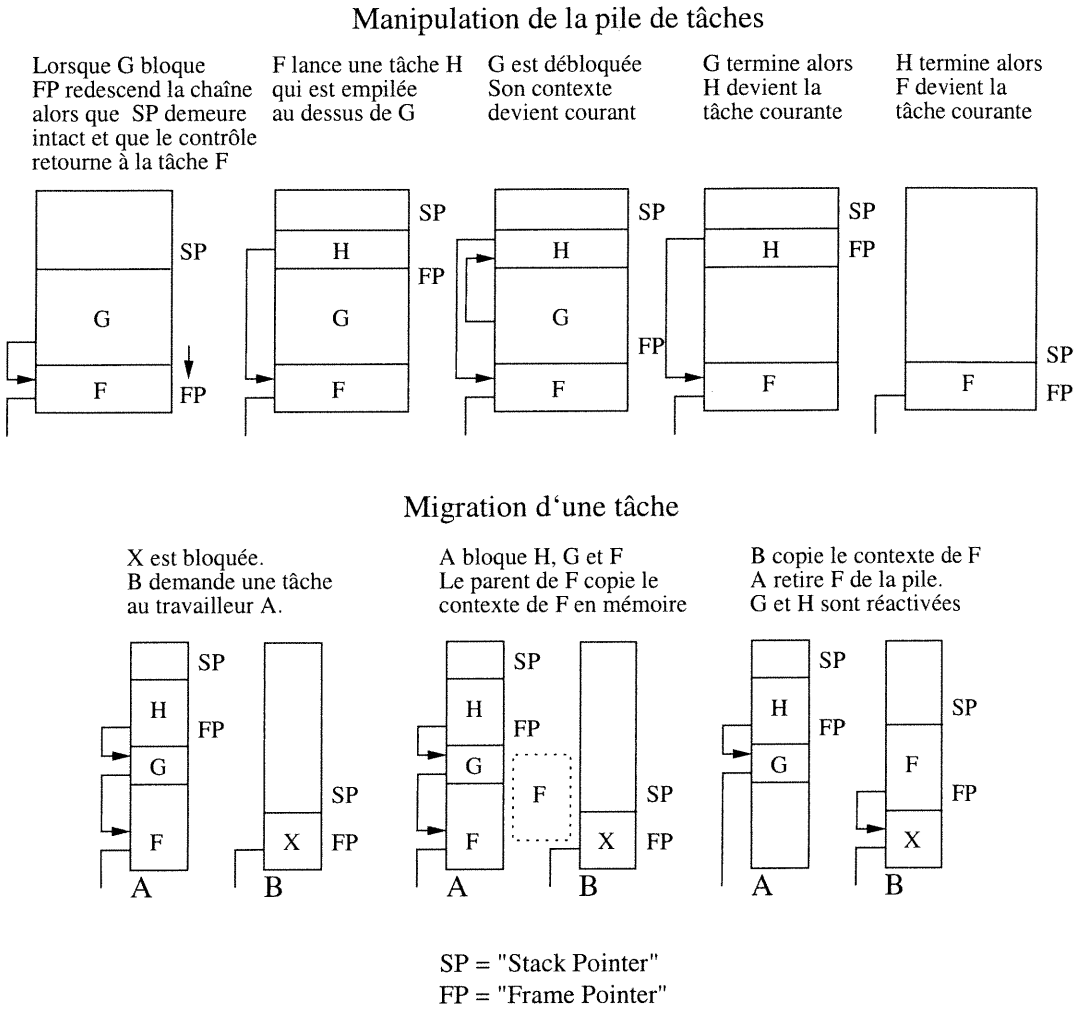


FIG. 2.5 – Le processus de manipulation des tâches de StackThreads

code assembleur est ensuite modifié par un script *awk* pour en retirer des informations intéressantes qui serviront à le modifier légèrement. Ainsi, toutes les primitives du langage s'utilisent comme des appels à des fonctions d'une bibliothèque, mais nécessitent toutefois une étape de transformation lors du processus de compilation.

Le fonctionnement interne de StackThreads est aussi très intéressant. Ainsi, chacun des processeurs logiques de StackThreads/MP emploie directement sa pile d'exécution à titre d'ensemble de tâches. L'ensemble de tâches consiste en une liste chaînée de contextes d'exécution. On peut accéder à cette liste à l'aide d'un pointeur qui se trouve dans le registre dénommé "frame pointer". La figure 2.5 illustre une séquence

d'évenements qui demandent la manipulation de cette liste. Dans cet exemple, le processeur exécute la fonction F. Arrive ensuite la construction `ST_THREAD_CREATE(G())`. Le processeur note alors l'existence d'un "fork" et exécute immédiatement l'appel à `G()`. Une telle approche permet à la création de tâches d'être très efficace car elle ne nécessite que 8 instructions de type RISC. Si la fonction G venait à bloquer (sur l'attente d'une mutex par exemple), alors son contexte serait retiré de la chaîne, tout en demeurant sur la pile d'exécution. Par la suite, le processeur remonte la chaîne de contextes pour trouver la prochaine tâche à exécuter. Dans ce cas-ci, c'est la fonction F. Supposons maintenant que F engendre un appel parallèle à H. Le processeur note à nouveau l'existence d'un "fork" et exécute l'appel à H. Pendant l'exécution de H, G pourrait débloquer. Ainsi, H serait bloquée et le contexte de G rattaché à la chaîne de contextes. G est maintenant la tâche courante. Notons cependant que le parent de G était F à l'origine et qu'il est devenu H. Ainsi, lorsque G termine son exécution, H continue la sienne. Finalement, lorsque H termine son exécution, F reprend le contrôle.

À son tour, le vol de tâches est implanté à l'aide d'une technique d'échantillonnage. Ainsi, lorsque la liste d'un processeur est vide ou lorsque les tâches qu'elle contient sont toutes bloquées, le processeur choisit une victime au hasard et lui demande une tâche. À la prochaine exécution de la macro `ST_POLLING` de la part de la victime, une tâche lui sera transférée (s'il y en a de disponible). De plus, cette migration de tâches s'effectue simplement à l'aide du blocage et du redémarrage des tâches. La figure 2.5 illustre le mécanisme de vol de tâche. Dans cet exemple, supposons que les tâches F, G et H se trouvent dans la liste du processeur A et qu'un autre processeur B fait une demande à A pour une tâche. Le processeur A commence d'abord par bloquer H et redémarre G (le parent de H), qui sera aussi immédiatement bloquée. On remonte ainsi jusqu'au parent de F (la tâche au début de la liste). Le parent de F copie ensuite le contexte d'exécution de F en mémoire partagée qui sera accessible par B. Il retire ensuite F de sa liste et redémarre G qui, à son tour, va redémarrer H.

Ainsi, pour migrer une tâche F, il est nécessaire de bloquer toutes les tâches de la pile

au dessus de F, ainsi que la tâche F. Il faut par la suite réactiver ces mêmes tâches en remontant la pile d'exécution. Le processus de vol de tâches est donc coûteux. Cependant, il est plus avantageux de procéder ainsi et de réduire la fréquence des vols que de transférer la tâche qui se trouve en fin de liste même si celle-ci est immédiatement accessible par le "frame pointer". En effet, lorsque l'arbre d'appels parallèles est relativement balancé, cette technique s'avère plus efficace car les tâches correspondant à des calculs plus gros sont transférées en premier réduisant ainsi la fréquence du vol de tâches. Comme pour CPar, l'approche préconise cependant un style de programmation "diviser pour régner".

Les tests de performances montrent que l'approche de StackThreads/MP est relativement efficace. Cependant, pour des fonctions d'une granularité aussi fine que la fonction de Fibonacci récursive, le surcoût de gestion est considérable. En effet, lorsque le programme à la figure 2.4 est exécuté sur un seul processeur, il est 2.5 fois plus lent que la fonction de Fibonacci récursive écrite en C [18].

Malgré l'originalité de l'implantation et les performances relativement bonnes de StackThreads, nous croyons qu'il y a place à l'amélioration. De plus, il serait préférable d'avoir un balancement de la charge qui soit entièrement automatisé. StackThreads/MP reste, à notre avis, une référence pour l'implantation des primitives de synchronisation.

2.6 Cilk

Développé par une équipe du MIT, Cilk 5.2 est un langage qui connaît un certain succès grâce à l'efficacité du code généré par son compilateur [3, 11]. Ce langage est une extension de C avec lequel CPar partage plusieurs caractéristiques, dont le balancement automatique de la charge. Cependant, Cilk se distingue sur plusieurs plans. D'abord, le système Cilk contient un ensemble d'outils pouvant faciliter le déverminage des programmes parallèles. Ensuite, parce que Cilk possède un vocabulaire plus riche que CPar, il supporte une gamme plus complète de styles de programmation parallèle. Finalement, on remarque une approche très différente dans la manière

d'implanter la création paresseuse des tâches malgré une très forte ressemblance dans l'implantation du vol de tâches.

Les outils de déverminage de Cilk permettent de détecter les situations de course et d'étreinte fatale. Rappelons qu'une situation de course survient lorsque plusieurs tâches manipulent une donnée commune et que le résultat du calcul dépend de l'ordonnancement des accès à cette donnée. L'étreinte fatale survient lorsqu'une ou plusieurs tâches attendent indéfiniment pour une ressource partagée. Dans la construction d'applications parallèles d'envergure, ce genre d'outil peut s'avérer très important. Ainsi, Cilk se classe aujourd'hui comme un langage de programmation parallèle de qualité "production", alors que ce n'est pas encore le cas pour CPar.

La figure 2.6 illustre le code Cilk pour la fonction de Fibonacci récursive. On constate que le lancement des tâches se fait simplement en plaçant le mot `spawn` devant un appel de fonction. La synchronisation est explicite et se fait à l'aide du mot `sync`. L'utilisation du mot réservé `sync` offre au programmeur une plus grande flexibilité en proposant un modèle de programmation plus général que celui de CPar. À l'opposé, la synchronisation dans CPar est implicite et a lieu immédiatement après le corps de la construction parallèle (voir la figure 1.4). La figure 2.7 illustre la différence conceptuelle entre ces deux modèles de synchronisation. Les "joins" implicites de CPar font converger deux flots de contrôle alors que ceux de Cilk, qui sont explicites, peuvent en faire converger un nombre arbitraire. La puissance expressive de Cilk n'est toutefois pas sans désavantages. En effet, elle a comme conséquence d'augmenter la complexité de la gestion des tâches et d'en réduire ainsi les performances brutes.

En plus des constructions de base `spawn` et `sync`, Cilk offre d'autres constructions telles les fonctions `inlet` et les variables qualifiées `private` [16]. Dans le premier cas, les fonctions `inlet` permettent de garantir l'atomicité d'une fonction relativement aux autres enfants d'une tâche. Dans le second, le mot `private` permet de déclarer une variable dont la valeur peut varier d'une tâche à l'autre (donc l'espace de stockage est différent pour chacune des tâches). Ce type de construction est utile dans un langage parallèle général. À ce stade, CPar ne contient aucune de ces

```
1  int fib(int n) {
2
3    if ( n < 2)
4      return 1 ;
5    else {
6      int f1, f2 ;
7      f1 = spawn fib(n-1) ;
8      f2 = spawn fib(n-2) ;
9      sync ;
10     return f1 + f2 ;
11   }
12 }
```

FIG. 2.6 – Exemple d'un programme Cilk

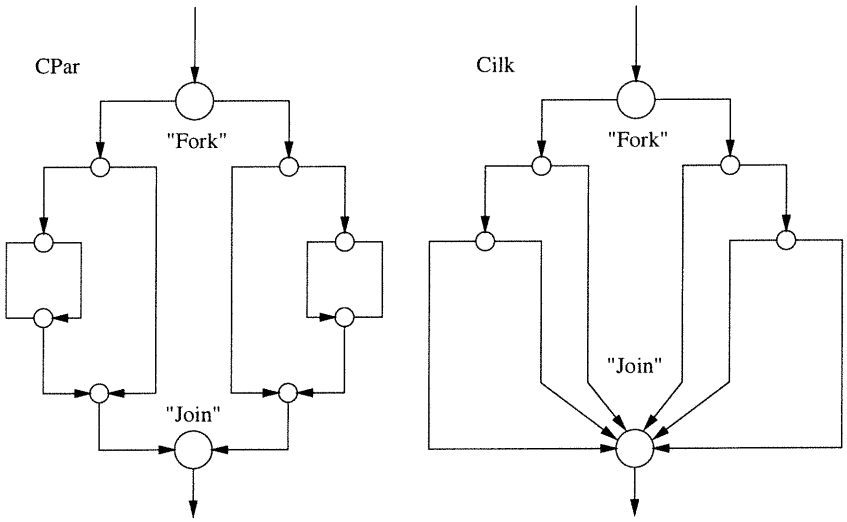


FIG. 2.7 – Les modèles de synchronisation de CPar et de Cilk

constructions supplémentaires, bien qu'il n'y ait pas de raison fondamentale qui les interdisent.

Le compilateur Cilk transforme le code source en un dialecte de C (celui de *gcc*). Les transformations appliquées sont relativement complexes. Toutes les fonctions faisant l'objet d'un appel parallèle sont dupliquées. La première fonction est dite "rapide" alors que l'autre est dite "lente". La fonction "lente" est celle qui sera exécutée par un voleur alors que la "rapide" est exécutée le reste du temps. Puisqu'en général une faible proportion des tâches sont volées, la grande majorité des tâches sont exécutées par l'entremise de la version "rapide". Cette technique peut cependant augmenter considérablement la taille du code exécutable. La figure 2.8 illustre le code C de la version "rapide" de la fonction de Fibonacci récursive après sa transformation.

Remarquons d'abord que les transformations ne sont pas locales : le code généré engendre un surcoût même aux feuilles de l'arbre d'appels (les lignes 9, 10, 11, 14) et où il n'y a aucun parallélisme possible. Considérant que le nombre de feuilles est généralement aussi grand que le nombre de nœuds dans le reste de l'arbre, le surcoût engendré est considérable. Ensuite, on peut voir que, lors de la création d'une tâche, Cilk doit initialiser une structure avec la valeur de toutes les variables locales vivantes et potentiellement modifiées. Cette approche possède donc un surcoût proportionnel à ce nombre de variables.

De la figure 2.8, on peut aussi observer que la structure interne de l'ensemble de tâches de chacun des processeurs est celle d'une pile et qu'elle est distincte de la pile d'exécution C. Cette approche offre l'avantage d'une plus grande portabilité au détriment d'un surcoût de gestion. On remarque aussi que lorsqu'un processeur rencontre un `spawn`, ce dernier ne crée pas une nouvelle tâche : il conserve l'état de la tâche courante (les lignes 20, 21, 27 et 28). En effet, plutôt que de créer une tâche, Cilk conserve dans une structure l'état de la tâche courante qui sera ensuite empilée sur sa pile. Le processeur peut ensuite effectuer l'appel de fonction du `spawn`. Il doit cependant vérifier à son retour que la tâche parent n'a pas été volée. Si tel est le cas, le processeur se met à rechercher du travail dans la pile de tâches des autres processeurs. La figure 2.9 illustre la structure interne de cette pile de tâches.

```
1  struct _fib_frame {
2    StackFrame header;
3    struct { int n; } scope0;
4    struct { int x, y; } scope1;
5  };
6
7  int fib(int n) {
8
9    struct _fib_frame * f;
10   f = alloc(sizeof(*f));
11   f->sig = fib_sig;
12
13   if (n < 2) {
14     free(f, sizeof(*f));
15     return n;
16   }
17   else {
18     int x, y;
19
20     f->header.entry = 1;
21     f->scope0.n = n;
22     push(f)
23     x = fib(n-1);
24     if ( pop(x) == FAILURE )
25       return 0;
26
27     f->entry = 1;
28     f->scope1.x = x;
29     push(f)
30     x = fib(n-2);
31     if ( pop(x) == FAILURE )
32       return 0;
33
34     free(f, sizeof(*f));
35     return (x+y);
36   }
37 }
```

FIG. 2.8 – La version rapide de Fibonacci après transformations

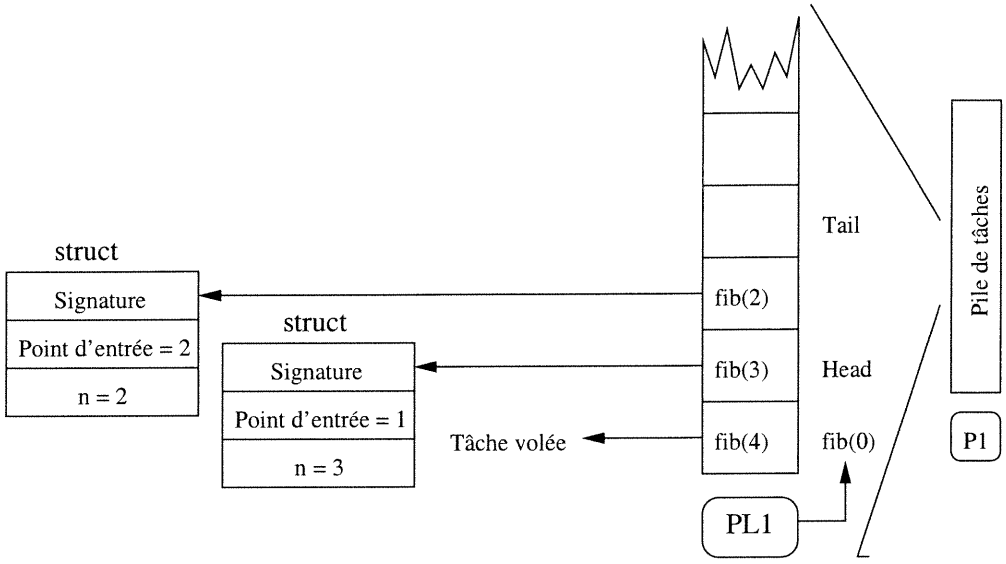


FIG. 2.9 – Structure interne de la pile de tâches Cilk

Le vol de tâches est implanté par l'entremise d'un protocole d'exclusion mutuelle qui possède une forte ressemblance à celui qu'emploie CPar. Alors que le mécanisme de synchronisation de CPar dépend des entrées de la pile de tâches, celui de Cilk repose plutôt sur les pointeurs de tête et de queue de cette pile. Cette approche comporte, comme nous le verrons, un désavantage important : les deux pointeurs doivent être accessibles à tous les processeurs et doivent donc se trouver en mémoire partagée. Par conséquent, l'empilement et le dépilement d'une tâche sont des opérations sensiblement plus coûteuses sur Cilk que sur CPar car le processeur ne peut conserver ses pointeurs dans sa cache. La figure 2.10 illustre un code simplifié du protocole d'exclusion mutuelle de Cilk. On remarque que l'empilement et le dépilement des tâches sont efficaces car aucun "lock" n'est effectué a moins que ce ne soit absolument nécessaire. De plus, comme pour StackThreads, le protocole prévoit que le vol soit effectué par le bas de la pile pour limiter la fréquence des vols et d'en augmenter leur "valeur" (la quantité de travail). Ainsi, comme StackThreads, l'implantation du vol de tâche privilégie un style de programmation "diviser pour régner".

Les performances de Cilk sont comparables à celles de StackThreads. Sur un seul processeur SPARC, le temps d'exécution de la fonction de Fibonacci récursive est environ 3 fois supérieur à l'exécution du programme C semblable. Sur la plateforme

Code de la victime

```
1  void push(frame *f) {
2    deque[T] = f;
3    T++;
4  }
5
6  void pop() {
7    T-;
8    if (H > T) {
9      T++;
10     Lock(L);
11     if (H > T) {
12       T++;
13       UnLock(L);
14       return FAILURE;
15     }
16     UnLock(L);
17   }
18   return SUCCESS;
19 }
```

Code du voleur

```
1  void steal() {
2    Lock(L);
3    H++;
4    if (H > T) {
5      H--;
6      UnLock(L);
7      return FAILURE;
8    }
9    UnLock(L);
10   return SUCCESS;
11 }
```

FIG. 2.10 – Le protocole d'exclusion mutuelle employé par Cilk

Intel on obtient un temps qui est 3.4 fois supérieur. Nous croyons que la force de Cilk réside surtout dans la richesse de ses constructions et des outils qu'il met à la disposition des programmeurs, mais qu'au niveau des performances, il y a place à amélioration surtout pour les programmes d'une granularité fine.

2.7 ParSubC

ParSubC [13] est une extension de C développée à l'Université de Montréal. Le langage, qui est le prédécesseur de CPar, se distingue de ce dernier de deux manières : d'une part, par sa capacité de distribuer le calcul sur un réseau d'ordinateurs et de l'autre, par sa technique de génération de code. ParSubC et CPar emploient donc les mêmes principes de création paresseuse de tâches et de vol de tâches malgré que ces techniques soient implantées différemment.

Le code à la figure 2.11 illustre bien l'utilisation de ParSubC. Comme on peut le constater, la syntaxe d'un appel parallèle est la même pour ParSubC que pour CPar à un détail près : le symbole `!!` remplace le mot réservé `par`. La sémantique d'un appel parallèle CPar reste la même que celui d'un appel parallèle de ParSubC. Par conséquent, l'appel de fonction peut être exécuté en parallèle avec le corps de l'expression qui suit le symbole `!!`.

```
1  int fib(int n) {
2    if ( n < 2)
3      return 1;
4    else {
5     int f1, f2;
6      f1 = fib(n-1) !! {
7        f2 = fib(n-2);
8      };
9      return f1 + f2;
10   }
11 }
```

FIG. 2.11 – Exemple d'un programme ParSubC

Un aspect intéressant de ParSubC est sa capacité de distribuer le calcul et ce, de manière transparente à l'usager. Ainsi, le langage offre au programmeur un modèle de programmation à mémoire partagée sur un réseau d'ordinateurs. La mémoire partagée est simulée à l'aide d'un gestionnaire de messages. Celui-ci distingue deux types de données : les variables globales et locales. Partant de l'hypothèse que les variables globales sont accédées en lecture beaucoup plus fréquemment qu'en écriture, une copie de ces dernières est conservée localement sur chacune des stations. La synchronisation (nécessaire) de ces copies se fait cependant à un coût élevé. À l'inverse, une variable locale peut être modifiée plus souvent. Ainsi, il est nécessaire de passer par le réseau en lecture comme en écriture, sauf pour la station qui la conserve localement. Comme cette variable sera généralement accédée par ce dernier, cette approche s'avère une bonne stratégie. Notons que ces problèmes ne se présentent que si le calcul est distribué.

De plus, parce qu'il permet de distribuer le calcul, ParSubC emploie un mécanisme d'échange de messages pour effectuer le vol de tâches. Ainsi, comme pour StackThreads, un processeur qui cherchait du travail devait formuler une demande de tâches à une victime. Comme l'envoi de messages devait être utilisé pour simuler la mémoire partagée, il était naturel d'utiliser la même approche pour implanter le vol de tâches. Celle-ci possède néanmoins une conséquence malheureuse pour un système parallèle : elle est coûteuse et réduit considérablement les performances.

Pour des raisons d'expérimentation et de prototypage, l'approche de génération de code est non conventionnelle. En effet, le compilateur modifie la totalité du code

source et le transforme en un code source en C qui agit à titre d'assembleur portable. En particulier, la pile d'exécution est gérée explicitement dans un tableau et le mécanisme de retour de fonction est basé sur l'utilisation d'un `switch` pour brancher au point de retour. La portabilité du code généré se fait donc au détriment d'une perte considérable de performances (mesurée au chapitre 4). De plus, la structure originale du code ayant disparu, le compilateur C ne peut effectuer aussi efficacement son travail d'optimisation.

Par conséquent, les performances des programmes exclusivement parallèles générés par le compilateur ParSubC ne sont pas très bonnes lorsqu'on les compare à celles de Cilk ou StackThreads. Cependant, les objectifs de ParSubC consistent à démontrer la faisabilité d'un tel compilateur. Ainsi, en employant les mêmes idées directrices, CPar cherche à améliorer considérablement les performances de ParSubC pour les applications parallèles.

2.8 CPar

Fondamentalement le même langage que ParSubC, CPar ne diffère que par ses objectifs et son implantation. À l'opposé de son prédécesseur, CPar ne permet pas de distribuer les calculs sur un réseau d'ordinateurs. Les recherches de L'Écuyer [13] démontrent que la distribution du calcul ne devient réalisable que lorsque la granularité des tâches est grande. Se voulant d'abord un langage pour le calcul parallèle à granularité fine, l'implantation actuelle du compilateur CPar, présentée dans ce mémoire, se concentre d'abord sur une parallélisation efficace pour un ordinateur à mémoire partagée. Soulignons que dans la suite de ce mémoire, nous ne distinguerons plus le langage de son implantation actuelle, que nous désignerons tous deux par le terme commun CPar.

Plusieurs aspects de CPar sont empruntés à d'autres langages. En les combinant et en les implantant d'une manière innovatrice, CPar tente d'obtenir un maximum de performances. CPar hérite de ParSubC les grandes idées directrices, qui sont : une syntaxe ainsi qu'un modèle de programmation simples, l'application de la création

paresseuse de tâches, ainsi que du vol de tâches, pour obtenir un balancement automatique de la charge. De plus, parce que CPar ne permet pas la distribution du calcul, il peut employer un protocole d'exclusion mutuelle plus proche de celui de Cilk pour augmenter ses performances.

Comme son prédécesseur, le compilateur CPar transforme aussi le code source en un code C portable (le dialecte de *gcc*). Le compilateur est un programme C qui effectue une lecture du programme, puis en construit une représentation arborescente. Il effectue ensuite les transformations appropriées avant de regénérer un nouveau programme. Cependant, les modifications apportées par le compilateur sont locales et n'affectent que les constructions parallèles. Un programme n'en contenant aucune reste donc inchangé après la compilation. De plus, il n'y a aucun surcoût engendré si le flot d'exécution ne passe pas par une construction parallèle. Une telle approche permet de conserver la structure originale du code tout en laissant au compilateur C la possibilité de faire son travail d'optimisation avec la même efficacité que sur le code d'origine séquentiel. La figure 2.12 illustre le résultat d'une compilation de la fonction de Fibonacci que l'on retrouve à la figure 1.4. Nous décrivons sommairement le code généré dans cette section. Le prochain chapitre donne les détails complets de fonctionnement.

Le code transformé commence d'abord par ajouter une déclaration de type aux lignes 1 à 5. Cette structure décrit simplement un appel parallèle à `fib`. On nomme cette structure "descripteur de tâche". Elle contient un pointeur vers une fonction (qui nécessite un "type cast"), un endroit pour y stocker le résultat de l'appel ainsi que la liste des arguments de l'appel. La ligne 7 déclare une fonction "proxy" qui sert à reconstruire l'appel original à partir du descripteur de tâche. À la ligne 11, le compilateur ajoute la déclaration du descripteur. Aux lignes 12 à 17, on retrouve le code de la fonction de Fibonacci tel qu'il est dans le programme original. Les lignes 18 à 20 illustrent la transformation que CPar applique à un appel parallèle. Plutôt que d'exécuter l'appel de fonction, il initialise les champs du descripteur et empile un pointeur vers celui-ci sur une pile de tâches ce qui le rend disponible à tous les autres processeurs. Enfin, le corps de l'expression parallèle est évalué aux

ligne 21 à 23. À la ligne 25 le processeur effectue la synchronisation implicite. Ainsi, il retire la tâche de sa pile et détermine s'il doit l'exécuter, attendre la fin de son exécution ou continuer le calcul en employant immédiatement le résultat qui est déjà disponible. Finalement, les lignes 30 à 33 décrivent la fonction "proxy" qui est capable de reconstruire l'appel parallèle à `fib` à l'aide d'un pointeur sur le descripteur. Cette fonction est employée par les processeurs qui exécutent des tâches pour le compte d'autres processeurs.

Comme on peut le constater, les modifications apportées au code source sont relativement simples. Puisque les déclarations de types n'engendrent aucune pénalité à l'exécution, seules les lignes 18, 19, 20 et la macro de la ligne 25 sont coûteuses, car elles sont systématiquement exécutées. À leur tour, les lignes 31 et 32 n'ont d'utilité qu'en cas de vol de tâche. Cet exemple nous permet aussi de mieux cibler nos efforts d'optimisation. Nous savons ainsi qu'il faut porter une attention particulière aux sections de code dont la fréquence d'exécution est élevée (les lignes 18, 19, 20 et 25).

L'objectif principal de CPar est de minimiser le surcoût engendré par l'utilisation du parallélisme. Alors que StackThreads et Cilk possèdent un surcoût de gestion d'environ 300% pour la fonction de Fibonacci sur une plateforme SPARC, CPar arrive à réduire la pénalité à moins de 40% environ pour cette même fonction (voir chapitre 4). Sur une plateforme Intel, le surcoût de gestion est réduit à moins de 5% pour des fonctions semblables et à 0% pour la fonction de Fibonacci en particulier. Pour atteindre ce niveau de performances, il s'avère nécessaire d'établir un modèle de concurrence, de développer des structures de données et des algorithmes bien adaptés à un environnement parallèle, ainsi que de prévoir des mécanismes de synchronisation efficaces. Finalement, seuls des tests empiriques variés pourront démontrer l'efficacité réelle des programmes CPar. Les chapitres qui suivent traitent des choix d'implantation de CPar et présentent une évaluation des performances du compilateur.

Notons que l'implantation de CPar est prévue pour une machine à mémoire partagée possédant un seul ou plusieurs processeurs. CPar dépend de la bibliothèque *pthread* (une implantation de Posix Threads) pour la gestion des processus légers et a été

```
 1 typedef struct {
 2   volatile void *_cparProxy ;
 3   volatile int _cparResult ;
 4   int _cparArg1 ;
 5 } _cparRpc0 ;
 6
 7 void _cparProxy0(_cparRpc0*) ;
 8
 9 int fib(int x) {
10
11 _cparRpc0 _cparDesc0 ;
12
13   int f1, f2 ;
14
15   if ( x < 2 )
16     return 1 ;
17
18 _cparDesc0._cparArg1 = x-1 ;
19 _cparDesc0._cparProxy = (void*) _cparProxy0 ;
20 _cparPush(_cparDesc0) ;
21                 {
22     f2 = fib(x-2) ;
23   } ;
24
25   f1 = _cparPop(_cparDesc0,
                   _cparDesc0._cparResult,
                   fib(_cparDesc0._cparArg1)) ;
26
27   return f1 + f2 ;
28 }
29
30 void _cparProxy0 (_cparRpc0 *_cparDesc) {
31   _cparDesc->_cparResult = fib(_cparDesc->_cparArg1) ;
32   _cparDesc->_cparProxy = NULL ;
33 }
```

FIG. 2.12 – Un exemple du code généré par le compilateur CPar

rédigé de manière portable. De plus, le noyau de CPar est conçu pour être le moins "vorace" possible lorsqu'il se trouve dans un état d'attente. Mentionnons que seules les transformations effectuées par le compilateur seront prises en compte dans ce mémoire. Les autres aspects tels l'analyse lexicale et syntaxique sont abondamment couverts dans d'autres références [10].

Chapitre 3

Implantation

3.1 Vue d'ensemble

Avant d'exposer les détails de CPar, il convient d'avoir une idée globale du fonctionnement d'un programme CPar. Par conséquent, nous allons simuler l'exécution d'un programme hypothétique afin de mettre en évidence certaines des composantes les plus importantes du noyau de CPar.

L'exécution de ce programme CPar commence tout d'abord par l'initialisation du noyau. De là sont lancés, sur des processus légers indépendants, un certain nombre de processeurs logiques. Ces processeurs se partagent les tâches du programme CPar qu'ils exécutent de manière séquentielle. Bien entendu, le nombre de processeurs logiques dépend du nombre de processeurs physiques, ou encore d'un paramètre spécifié à la ligne de commande. En général, plus il y a de processeurs, plus l'exécution de notre programme sera rapide même s'il n'y a généralement que peu d'avantages à avoir plus de processeurs logiques qu'il n'y a de processeurs physiques.

Idéalement, chacun des processeurs logiques ("thread") devrait être exécuté sur un processeur physique distinct. Nous laissons cependant le soin au système d'exploitation d'en faire la répartition sur l'ensemble des processeurs de l'ordinateur. Tous les processeurs, à l'exception d'un seul, suivent la même logique. Les états possibles de chaque processeur sont représentés graphiquement à la figure 3.1. L'exception

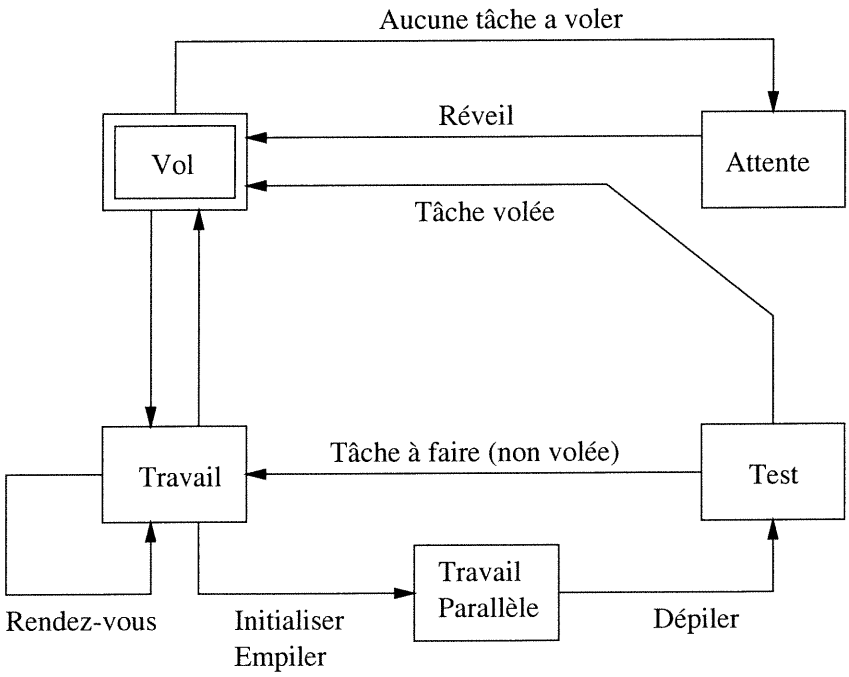


FIG. 3.1 – Les états d'un processeur

demeure le processeur logique principal, celui qui exécute la fonction "main" du programme. Il ne diffère que par son état initial : il commence à l'état "travail" plutôt que celui de "vol". Dans tous les cas, la stratégie employée est simple :

```
1    Tant que le programme n'est pas terminé
2        Voler une tâche Y à un autre processeur
3        Exécuter la tâche Y
```

Lors de l'exécution d'une tâche, il se peut que le processeur ait à lancer des tâches parallèles. C'est alors qu'il exécutera le pseudocode suivant :

```
1    Lancer une tâche X en parallèle correspondant à l'appel de fonction
2
3     Exécuter le bloc d'énoncés
4
5     Si la tâche X est volée alors
6         Tant que la tâche X n'est pas terminée
7             Voler une tâche Y à un autre processeur
8             Exécuter la tâche Y
9     Sinon
10        Exécuter la tâche X
```

Cette simulation suggère qu'un processus léger n'est pas initié pour chaque tâche créée par un programme CPar. Cette supposition s'avère correcte car dans le cas

contraire, le nombre de processus légers dépasserait rapidement le nombre de processeurs physiques et surchargerait ainsi le système d'exploitation. La création d'une tâche parallèle en est donc réduite à la manipulation d'une structure de données par l'un des processeurs. Ces structures étant simultanément disponibles aux autres processeurs, par l'entremise d'une structure globale distribuée, permettent le travail en parallèle et le balancement automatique de la charge. Afin de minimiser l'impact de toute cette gestion, il faut cependant développer des structures de données et des mécanismes de synchronisation simples, mais efficaces.

3.2 Modèle de concurrence

CPar ne s'occupe pas de la gestion des processus légers. A cette fin, il utilise la bibliothèque *pthreads* qui est une implantation du standard Posix Threads. Cette bibliothèque offre l'avantage de la portabilité car elle est disponible sur pratiquement toutes les plateformes. En revanche, elle ne mentionne pas, ou peu, de détails quant à son implantation. Plus particulièrement, elle ne spécifie pas les modèles de concurrence que la bibliothèque doit supporter.

On entend par un modèle de concurrence la relation entre le nombre de processus légers et le nombre d'entités systèmes (une structure gérée par le système d'exploitation qui reçoit une fraction du temps UCT). De ce point de vue, les implantations de *pthreads* diffèrent d'une plateforme à l'autre. Certaines utilisent un modèle N à 1, d'autres 1 à 1 ou encore N à M [4]. Quelques observations sur ces trois modèles permettront de déterminer lequel d'entre eux correspond à nos besoins.

Le modèle N à 1. Ici, les ressources d'une entité système sont partagées parmi tous les processus légers d'un programme. Il revient donc à la bibliothèque qui les implante d'en faire la gestion. Cependant, puisque ce type de concurrence ne permet pas d'utiliser plus d'un processeur à la fois (pour un programme donné), on ne peut pas tirer avantage d'un ordinateur multiprocesseurs. Par conséquent, ce type de concurrence ne peut pas être employé par le noyau de CPar.

Le modèle 1 à 1. Ce modèle de concurrence s'avère le plus simple car à chaque processus léger est associée une entité système dédiée. Il permet donc d'employer toutes les ressources de calcul d'une machine. Néanmoins, une utilisation trop libérale des processus légers peut surcharger le système d'exploitation. En particulier, lorsque leur nombre dépasse considérablement le nombre de processeurs physiques, le système d'exploitation doit alors consacrer de plus en plus de temps à leur gestion. Comme CPar utilise les processus légers pour le lancement des processeurs et que leur nombre est généralement inférieur ou égal au nombre de processeurs physiques, ce modèle de concurrence convient parfaitement. Notons que l'implantation de *pthreads* sur Linux ne supporte que ce type de concurrence.

Le modèle N à M. Ce modèle tente de réconcilier les deux premiers en y incorporant le meilleur de chacun. En effet, celui-ci permet de conserver une partie des processus légers sous le contrôle d'une bibliothèque, tout en mettant à la disposition de l'usager autant d'entités système que ce dernier le désire. Le modèle 1 à 1 est donc un cas particulier du modèle N à M. Malgré une complexité supérieure, ce modèle offre la plus grande flexibilité. L'implantation *pthreads* sur Solaris supporte d'ailleurs seulement ce type de concurrence.

Comme nous désirons que CPar soit portable, nous n'avons d'autre choix que d'employer le modèle de concurrence 1 à 1. Il nous faudra par conséquent adapter notre noyau dans le cas où l'implantation *pthreads* utilisée ne supporte que le modèle N à M. Il existe cependant un autre avantage important à utiliser ce type de concurrence. En effet, comme le système d'exploitation est en mesure de faire un choix globalement bon quant à l'emplacement (sur quel processeur physique) d'un processeur logique, il est prudent de lui laisser prendre cette décision afin d'obtenir un balancement efficace de la charge.

3.3 La pile de tâches de CPar

Pour permettre un balancement automatique de la charge, toutes les tâches doivent être accessibles par tous les processeurs. Une pile globale de tâches nécessiterait

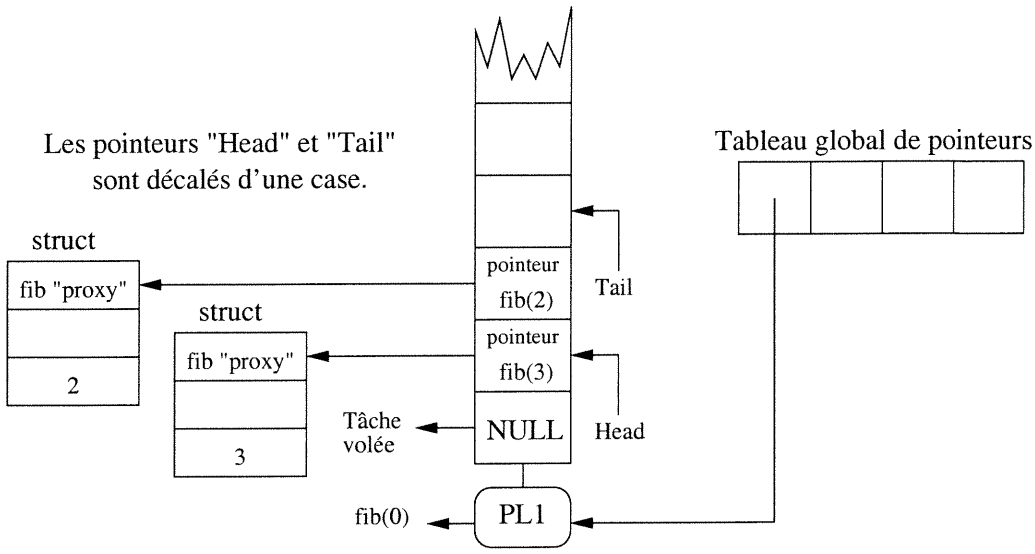


FIG. 3.2 – La structure interne de la pile de tâches de CPar

une synchronisation excessive qui réduirait considérablement les performances de CPar. Par conséquent, comme Cilk et ParSubC, chaque processeur possède sa propre pile locale de tâches. Les piles sont partagées par l'entremise d'un tableau global de pointeurs. On évite ainsi une synchronisaton trop fréquente et on obtient de meilleures performances.

Comme ces deux autres langages, la pile de tâches de CPar est caractérisée par deux pointeurs : "head" et "tail". Le premier indique l'emplacement de la prochaine tâche à être volée, alors que le second correspond au pointeur du dessus de la pile. La figure 3.2 illustre la structure interne de la pile de tâches de CPar. On distingue d'abord l'existence d'un tableau global qui donne accès à toutes les piles de tâches. On remarque ensuite que la pile de tâches est un tableau de pointeurs vers des structures que l'on nomme "descripteurs de tâches". Ce sont ces structures de données qui sont manipulées par les processeurs logiques afin d'éviter la création d'un grand nombre de processus légers.

Néanmoins, comme dans toute structure partagée, cette pile distribuée amène un problème de synchronisation. En effet, pour que le calcul soit exact, chaque tâche ne peut être exécutée qu'une seule fois. Il est donc important qu'un seul processeur à la fois, qu'il soit une victime ou un voleur, ait accès à une tâche donnée. Plu-

sieurs approches sont possibles pour synchroniser l'accès à une pile de tâches : les interruptions, l'échange de messages et un protocole d'exclusion mutuelle n'en sont que trois. L'emploi des interruptions s'avère complexe et lent. Celui de l'échange de messages, quoique plus rapide que les interruptions, demeure toutefois mieux adapté à un environnement distribué qu'à un environnement parallèle à mémoire partagée. CPar emploie plutôt la dernière approche et implante une légère variante du protocole d'exclusion mutuelle proposé dans la thèse de Feeley [9]. Dans un contexte à mémoire partagée, ce protocole, présenté et formellement prouvé par Feeley, procure des avantages qui s'avèrent considérables. La figure 3.3 illustre l'algorithme dans le contexte du vol de tâches.

Le code dans la colonne de gauche est exécuté par tous les processeurs lorsqu'ils veulent retirer une tâche de leur pile pour l'exécuter. Dans le cadre du vol d'une tâche, on appelle ces processeurs les "victimes". La colonne de droite contient le code exécuté par un processeur qui veut "voler" une tâche à la victime. Notons que la partie de gauche est exécutée à tous les sites d'appels parallèles, alors que la partie de droite ne l'est que lorsqu'un processeur veut voler une tâche à un autre. Ainsi, le code de la colonne de gauche est exécuté avec une fréquence beaucoup plus importante que le code dans la colonne de droite. Il est donc essentiel que le code exécuté par la victime soit le plus efficace possible.

La victime, comme le voleur, ne peuvent s'approprier une tâche de la pile que si cette dernière en contient. Ainsi, la pile de tâches est non-vide si `head > tail` ; autrement, la pile est vide et `head == tail`. De plus, un invariant est maintenu dans le tableau qui implante cette pile : `tail[i] == NULL` lorsque `i > 0`.

Ainsi, lorsqu'une victime veut s'approprier une des tâches, elle n'a qu'a vérifier que la pile ne soit pas vide (colonne gauche à la ligne 5). S'il ne restait qu'une tâche dans la pile alors il est possible qu'une situation de course survienent. Dans ce cas, ce sont les lignes 7 à 15 qui détermineront qui pourra exécuter la tâche. Le code de la victime pour retirer une tâche de sa pile est très efficace car, habituellement, seules les lignes 3, 5 et 19 sont exécutées.

Le voleur doit aussi s'assurer que la pile de la victime contienne au moins une tâche

avant de la voler. Ce test pourrait être fait de la même manière que pour la victime. Cependant, nous constaterons qu'il est avantageux de ne pas laisser au voleur l'accès à la variable "tail". Nous devons alors nous fier à l'invariant de la pile (colonne de droite à ligne 3). Ainsi, lorsque l'expression `head[1] != NULL` est vrai, il y doit y avoir au moins une tâche sur la pile et le voleur tente de l'obtenir.

Remarquons maintenant que le processeur qui tente de voler une tâche à un autre n'a jamais besoin de référencer la variable "tail" de la victime. Cette observation est à la base d'une méthode d'accès efficace à la pile de tâches. Le protocole suppose cependant que les écritures en mémoire soient de type "write-through", qu'elles affectent directement la mémoire centrale de l'ordinateur, et donc que la nouvelle valeur soit immédiatement accessible aux autres processeurs. Or, les processeurs modernes sont équipés de mémoire cache et emploient plutôt des écritures en mémoire de type "copy-back". Celles-ci modifient une valeur dans la mémoire cache seulement, et ce n'est qu'après un certain délai qu'elle sera propagée à la mémoire centrale. Dans le cas particulier de ce protocole d'exclusion, il est impératif que l'assignation à la ligne 3 de la fonction `attempt_pop` (colonne de gauche) soit immédiatement visible des autres processeurs car elle affecte l'objet de la synchronisation. Il faut donc employer une écriture de type "write-through" à cet endroit. Ce type d'écriture demande cependant beaucoup plus d'effort de la part d'un processeur (de un à trois ordres de grandeur en termes de cycle machine). L'opération est d'autant plus coûteuse qu'elle est effectuée à chaque fois qu'un processeur veut exécuter une tâche. Nous verrons néanmoins qu'il existe un moyen d'éviter cette inefficacité en relaxant le protocole d'exclusion mutuelle.

3.4 L'accès à la pile de tâches

Un accès très rapide aux données locales d'un processeur est un aspect essentiel aux performances générales de CPar. Cette rapidité est d'autant plus importante lorsqu'on accède fréquemment à une donnée. À partir du protocole d'exclusion mutuelle illustré à la figure 3.3, il est possible de dresser une liste des opérations, ainsi que de leurs fréquences, effectuées sur les principales variables locales d'un processeur. Le

```
1  STATUS attempt_pop() {
2
3    *tail-- = NULL ;
4
5    if (head > tail) {
6
7      boolean is_stolen ;
8
9      acquire_lock(head_lock) ;
10     is_stolen = (head > tail) ;
11     release_unlock(head_lock) ;
12
13     if ( is_stolen ) {
14       tail++ ;
15       return failure ;
16     }
17   }
18
19   return success ;
20 }
```

```
1  TASK_DESC attempt_steal() {
2
3    if ( head[1] != NULL ) {
4
5      task_desc_t *desc ;
6
7      acquire_lock(head_lock) ;
8      head++ ;
9      desc = *head ;
10
11     if ( desc == NULL )
12       head-- ;
13
14     release_lock(head_lock) ;
15
16     return desc ;
17   }
18
19   return NULL ;
20 }
```

FIG. 3.3 – Le protocole d'exclusion mutuelle de CPar

Variable	Opération	Processeur	Autres processeurs	Fréquence
tail	lecture	oui	non	élevée
	empiler	oui	non	élevée
	dépiler	oui	non	élevée
head	lecture	oui	oui	basse
	écriture	oui	oui	basse
autres	lecture	oui	oui	basse

TAB. 3.1 – Opérations sur les variables locales d'un processeur

tableau 3.1 en fait l'énumération.

On remarque que le pointeur `tail`, qui représente le dessus de la pile de tâches du processeur, est la variable le plus souvent accédée par le processeur. Il est donc très important de pouvoir y accéder à un coût minimal. Notons que cette dernière ne peut être lue par d'autres processeurs et qu'elle n'est modifiée que par son propriétaire. Ainsi, nous pouvons mettre au point une stratégie très efficace d'accès aux données. De toutes les approches envisagées, une seule possède les caractéristiques désirées.

3.4.1 Une approche portable

Notre noyau repose sur la bibliothèque standard *pthreads*, disponible avec toutes les distributions de *Linux*, qui prévoit la possibilité de créer des données locales pour chacun des processeurs grâce à "Thread Specific Data". À chaque donnée locale est associée une clé qui permet de retrouver la donnée sur demande. Cette clé est en

pratique un index dans un tableau. Cependant, la fonction qui permet de retrouver la donnée nécessite un appel de fonction de plusieurs dizaines d'instructions. Il est évident que dans un programme d'une granularité fine, une telle façon de procéder engendre à l'exécution un coût beaucoup trop élevé.

En effet, chaque accès (en lecture ou en écriture) d'une donnée locale requiert l'exécution de deux appels de fonction. Pour une donnée dont la fréquence d'accès est aussi importante que celle du pointeur sur le dessus de la pile de tâches, cette approche est inacceptable. De plus, pour une fonction à granularité très fine, le coût relatif de ces appels de fonction peut s'avérer plus coûteux que le corps même de la fonction. Il existe cependant un moyen d'améliorer cette technique.

Ainsi, si l'unique donnée locale est un pointeur vers une structure globale contenant toutes les autres données locales, le nombre d'appels peut être réduit à un par fonction. Puisque le pointeur vers la structure est une constante, il n'est jamais question d'écriture directe mais seulement d'une lecture à l'entrée d'une fonction. Une indirection étant plus efficace qu'un appel de fonction, le gain de performance serait considérable. Malgré tout, l'appel à `__pthread_getspecific` au début d'une fonction engendre un surcoût qui demeure trop élevé pour une fonction d'une granularité fine.

Il serait certainement possible d'étendre la structure interne d'un processus léger afin d'y ajouter un pointeur. Ce pointeur pourrait permettre l'accès rapide à une structure contenant toutes les variables propres à un processeur. Cependant, cette approche possède un inconvénient important. L'extension d'une bibliothèque nous lie définitivement à une implantation spécifique de *pthreads* nuisant ainsi considérablement à la portabilité du noyau. Puisque la portabilité est un de nos objectifs, cette solution s'avère sans intérêt.

3.4.2 Une technique plus efficace

Les données locales qui sont le plus facilement accessibles d'un processeur sont ses registres. Il serait donc intéressant de réserver l'utilisation de l'un d'eux pour contenir un pointeur vers les données locales d'un processeur. De plus, comme le système d'exploitation sauvegarde l'ensemble des registres lors d'un changement de contexte,

la technique peut fonctionner même si le nombre de processeurs logiques est supérieur au nombre de processeurs physiques.

Cependant, cette technique possède deux prérequis. D'abord, il est nécessaire d'avoir un support syntaxique de la part du compilateur C, et ensuite, de déterminer ce que doit contenir ce registre. Car un pointeur vers une structure globale ne s'avère pas forcément la solution la plus avantageuse. L'aspect syntaxique est immédiatement résolu par l'emploi du compilateur *gcc*. Celui-ci offre une extension qui permet de déclarer une variable globale dont la valeur se trouve dans un registre spécifique. Malgré la perte de portabilité engendrée par l'emploi de cette extension, le compilateur *gcc* est suffisamment répandu pour justifier le gain potentiel de performance. Enfin, pour déterminer le contenu du registre, nous faisons appel au tableau 3.1 qui énumère les opérations possibles sur les variables locales d'un processeur logique.

À priori, la solution la plus efficace consiste à inclure dans ce registre un pointeur vers une structure globale qui contiendrait toutes les données locales d'un processeur. Appelons ce pointeur "pLoc". Ainsi toutes les données locales doivent être accédées via une indirection de "pLoc". La figure 3.4 illustre le code assembleur optimisé tel que généré par le compilateur *gcc* pour chacune des opérations énumérées à la figure 3.1. Comme on peut le remarquer, toutes les opérations nécessitent un accès à la mémoire. Certaines opérations fréquentes, comme l'empilement et le dépilement d'une tâche, engendrent trois instructions et autant d'accès à la mémoire. Il serait intéressant de pouvoir réduire au minimum le nombre d'accès en mémoire.

La figure 3.1 nous indique que les opérations les plus fréquentes sont effectuées sur la variable locale "tail" qui constitue le pointeur sur le dessus de la pile de tâches CPar. De plus, comme nous l'indique cette même figure, seul le processeur "propriétaire" de la variable "tail" peut y faire référence. On peut se demander s'il est plus avantageux de conserver ce pointeur dans un registre plutôt qu'un pointeur sur une structure globale. Ce serait certainement le cas si nous arrivions à retrouver aisément un pointeur vers une structure contenant le reste des données locales. Pour y parvenir, nous proposons une solution simple et peu coûteuse.

En supposant que la structure contenant les données locales soit allouée sur une

Variable	Opération	Code assembleur optimisé	Instr.	Ref. mém.
tail	lecture	`mov eax, [offset](ebx)`	1	1

	empiler	`mov eax, [offset](ebx)`
		`mov [offset](eax),pDescr`
		`add [offset](ebx),-4`
			3	3

	dépiler	`mov eax, [offset](ebx)`
		`mov [-4](eax), 0`
		`add [offset](ebx), 4`
			3	3

head	lecture	Voir tail	3	3
	dépiler	Voir tail	3	3
autre	lecture	`mov eax, [offset](ebx)`	1	1

Code pour un Pentium (Intel) et où le registre ebx est réservé par CPar.
On suppose que pDesc se trouve déjà dans un registre au moment des opérations.

FIG. 3.4 – Code généré lorsque le registre contient l'adresse d'une structure

Variable	Opération	Code assembleur optimisé	Instr.	Ref. mém.
tail	lecture	Aucun code généré !	0	0
	empiler	`mov -4(ebx), pDescr` `add ebx, -4`	2	1
	dépiler	`mov (ebx), 0` `add ebx, 4`	2	1
head	lecture	`Voir "autre"`	3	1

	dépiler	`mov eax, ebx`
		`and eax, 0xfffff000`
		`add [offset](eax), 4`
			3	1

autre	lecture	`mov eax, ebx`
		`and eax, 0xfffff000`
		`mov eax, [offset](eax)`
			3	1

Code pour un Pentium (Intel) et où le registre ebx est réservé par CPar.
On suppose que pDesc se trouve déjà dans un registre au moment des opérations.

FIG. 3.5 – Code généré lorsque le registre contient le pointeur "tail"

frontière de 4Ko (par exemple), une simple opération logique sur le pointeur "tail", qui pointe dans un espace intérieur à la structure, nous permet d'obtenir un pointeur sur la structure entière. De plus, une opération logique entre deux registres étant très peu coûteuse, l'opération n'engendre que peu de pertes de performances. Comme on peut le remarquer sur les sections de codes à la figure 3.5, cette approche génère un code plus compact et plus efficace que la première alternative. En effet, on remarque maintenant que les opérations les plus fréquentes nécessitent moins d'instructions et moins d'accès à la mémoire. En particulier, la lecture du pointeur "tail" n'engendre aucun code particulier puisqu'il se trouve déjà dans un registre.

L'emploi d'un registre dédié pour accéder aux données locales d'un processeur nous permet d'obtenir un gain de performance important. Ce gain n'est cependant pas

sans inconvénients. D'abord, il faut noter que sur certains processeurs (notamment Intel), les registres sont déjà peu nombreux à l'origine. Monopoliser l'un d'entre eux peut réduire les possibilités d'optimisation de la part du compilateur et contribuer ainsi à une perte de performances.

De plus, dans un contexte de compilation séparée, il est impossible d'appeler une fonction parallélisée d'un module qui n'a pas été compilé avec CPar, ou encore avec une option à la ligne de commande spécifiant qu'un registre particulier était réservé. Cette limitation provient de ce que le module "non-CPar" n'est pas conscient du contenu du registre réservé. Finalement, cette approche nécessite un compilateur qui supporte l'allocation globale des registres. Étant donné que les noms des registres varient d'une plateforme à l'autre, cela pose aussi un problème de portabilité. Pour contrer ces derniers inconvénients, toutes les fonctions non portables du système sont localisées dans un seul fichier. Ainsi, le noyau de CPar demeure très performant tout en étant quasi portable.

Notons aussi que l'architecture SPARC offre au programmeur jusqu'à sept registres globaux. Le compilateur *gcc* nous permet d'en employer deux. Par conséquent, un deuxième registre sert à contenir l'adresse de la structure globale. Cela nous permet d'économiser une instruction à chaque fois que l'on fait référence à un champ de cette structure.

3.5 Le descripteur de tâche

La sémantique que nous donnons à un appel parallèle ressemble beaucoup à celle d'un "Remote Procedure Call" (RPC). RPC est un protocole synchrone que l'on retrouve dans les architectures client/serveur. Il permet à un programme de faire appel aux services d'un autre programme habituellement situé sur une autre machine et ce, sans le souci des détails de la communication réseau. Ce choix de sémantique est délibéré, car il rend l'interprétation d'une construction parallèle facile et intuitive. De plus, parce que le protocole n'est pas bloquant (même s'il est synchrone), il est donc possible d'effectuer un travail utile lorsqu'on est en attente.

Le protocole RPC ressemble sensiblement à un appel de procédure normal. Lors d'un appel de procédure local, le client place les arguments à un endroit spécifique connu (la pile ou les registres) du client et du serveur (la fonction), transfert le contrôle à la procédure appelée puis, lorsque l'appel est terminé, le client reprend le contrôle et peut en extraire le résultat, qui se trouve lui aussi à un endroit connu. Pour un appel RPC, les arguments, ainsi que le nom de la fonction à être exécutée, sont placés dans une structure (que l'on nomme descripteur) connue des deux partis. On procède aussi de la même façon pour un appel parallèle.

Le descripteur de tâche doit simplement contenir suffisamment d'informations pour que l'appel de fonction puisse être reconstruit. On sait qu'il faut y stocker au minimum le nom de la fonction à exécuter, un nombre potentiellement variable d'arguments et, si nécessaire, l'espace pour le résultat du calcul. Dans un environnement concurrent, il faut aussi un moyen de détecter la fin de l'appel parallèle. Ainsi, pour le nom de la fonction, on peut utiliser son adresse, suivi d'un champ pour chaque argument de l'appel. Si nécessaire, un champ supplémentaire peut être ajouté afin de conserver le résultat de l'appel parallèle. Enfin, nous verrons que le premier mot machine du descripteur, peut aussi permettre de détecter la fin d'un calcul parallèle. Les lignes 1 à 5 de la figure 2.12 (à la page 37) illustrent le descripteur généré pour un appel à la fonction `fib`.

Dans cet exemple, le champ `_cparProxy` sert à conserver l'adresse de la fonction qui se trouve à la ligne 30. Le champ `_cparResult` contient le résultat d'un appel lorsque celui-ci a été volé. Enfin, `_cparArg1` contient l'unique argument de la fontion `fib`. Notons que dans cet exemple, la fonction appelée est connue à la compilation. Dans le cas contraire, il y aurait un champ supplémentaire `_cparFun` qui contiendrait une autre adresse de fonction. Cette dernière serait employée par la fonction "proxy" pour effectuer l'appel de fonction approprié.

Néanmoins, un RPC et un appel parallèle se distinguent par leur point de synchronisation respectif. Puisque le protocole RPC est synchrone, le point de synchronisation est donc implicite et se trouve au site même de l'appel. Or, dans la construction parallèle de CPar, il se trouve à la fin de la construction, ce qui laisse la possibilité

d'exécuter d'autres instructions en parallèle avant le point de synchronisation.

3.6 Les fonctions "proxy"

Le descripteur de tâche sert à reconstruire un appel de fonction. Puisque tous les appels parallèles ne possèdent pas le même nombre d'arguments et n'appellent pas la même fonction cible, ils ne peuvent être traités efficacement par une seule structure de donnée ni par une fonction générique de reconstruction. L'emploi de fonctions "proxy" apporte une réponse simple et peu coûteuse à ce problème.

Essentiellement, une fonction "proxy" est une fonction à un seul argument, un pointeur vers un descripteur, sachant reconstruire l'appel vers la fonction cible à partir de ce descripteur. Si l'on convient de placer l'adresse de la fonction "proxy" au début du descripteur de tâche, l'interface d'exécution devient alors uniforme et très simple. En effet, il suffit d'effectuer l'appel suivant pour exécuter une tâche volée : `(*desc)(desc)`. Il suffit uniquement d'avoir un pointeur vers son descripteur pour exécuter une tâche. Les lignes 30 à 33 de la figure 2.12 (à la page 37) fournissent un bon exemple d'une fonction "proxy".

Si nous utilisons les fonctions "proxy", c'est pour éviter d'avoir à modifier le code original plus qu'il n'est nécessaire et de réduire ainsi l'impact de la parallélisation sur la taille du programme. En effet, la fonction "proxy" pourrait être évitée si chaque procédure faisant l'objet d'un appel parallèle était doublée, tout en conservant comme seul argument un descripteur de tâche. Cependant, il n'est pas certain de pouvoir obtenir un gain considérable en vitesse avec une telle approche. De plus, cette dernière peut accroître considérablement la taille du code compilé.

L'approche ouvre aussi un potentiel d'optimisation. Ainsi, lorsqu'un processeur veut exécuter une tâche de sa propre pile, il lui est possible de "court-circuiter" la fonction "proxy" et d'appeler directement la fonction cible car elle lui est connue. En effet, le compilateur peut générer directement l'appel de la fonction avec les arguments qui se trouvent dans le descripteur, économisant ainsi le surcoût d'un appel de fonction. On peut dire alors que le processeur "court-circuite" l'appel de fonction "proxy" en

procédant ainsi :

`(*desc)(desc)` devient `desc.fn(desc.arg1, ...)`

Comme il s'agit du cas le plus fréquent lorsque le nombre de tâches est important, l'économie est substantielle car elle évite un très grand nombre d'appels de fonction inutiles. Notons cependant que si le processeur n'a pas créé la tâche, il doit nécessairement passer par la fonction "proxy". Puisqu'il n'est pas en mesure d'interpréter le contenu du descripteur ou de connaître la fonction cible, ce sera à la fonction "proxy" de le faire à sa place. Ce qui permet à un processeur de prendre avantage d'une telle optimisation, c'est qu'il possède toute l'information de type nécessaire, car c'est lui-même qui a fait l'allocation du descripteur au niveau du code source.

3.7 L'allocation des descripteurs sur la pile

Il demeure certain que la seule façon vraiment efficace d'effectuer l'allocation d'un descripteur (et donc d'une tâche), c'est de le faire sur la pile d'exécution. En effet, n'importe quel appel de fonction (par exemple `malloc`) génère un coût beaucoup trop élevé à l'exécution. L'allocation d'une tâche est une activité à très haute fréquence. Par conséquent, nous devons l'optimiser au maximum.

Pour CPar, le coût de l'ensemble des allocations de descripteurs dans le corps d'une fonction est d'au plus une seule instruction assembleur. En effet, les descripteurs sont dans les faits des variables locales à chaque fonction. Si cette dernière ne possédait pas déjà de variables locales, il y aurait un surcoût d'une instruction pour déplacer le pointeur de pile. Si au contraire elle en avait, l'approche n'engendrerait aucun surcoût. L'allocation peut alors être considérée peu coûteuse, voire même optimale.

3.8 Initialisation du descripteur

L'initialisation d'un descripteur est une opération coûteuse. Elle peut même être la cause de la quasi totalité du surcoût de gestion d'un programme CPar. Il est donc

très important d'en minimiser l'impact. L'approche que CPar emploie est plutôt simple. Elle s'avère néanmoins flexible et offre plusieurs possibilités d'optimisation. Plus particulièrement, elle se prête bien à la propagation des constantes. De plus, dans certaines circonstances, il est possible de prendre avantage de la convention du passage de paramètres pour réduire davantage le surcoût d'initialisation.

D'abord, comme l'illustre le code de la figure 2.12 à la page 37, chaque paramètre de l'appel de fonction est copié dans un des champs du descripteur. Ensuite, il est nécessaire de copier l'adresse de la fonction, ainsi que celle de la fonction "proxy". Une opportunité d'optimisation se présente lorsque certains champs du descripteur sont constants. On peut effectivement omettre de les stocker s'ils sont propagés à l'intérieur de la fonction "proxy" et sur l'appel qui la "court-circuite". Par exemple, dans le cas où le nom de la fonction cible est constant, l'appel peut être optimisé en ne stockant pas l'adresse de celle-ci et en propageant son nom dans la fonction "proxy". Comme on peut le remarquer à la figure 2.12, l'optimisation permet d'éliminer l'adresse de la fonction (`fib` dans ce cas) du descripteur, ainsi que son initialisation. On retrouve alors un appel direct à `fib` dans la fonction "proxy" ainsi que sur l'appel qui la "court-circuite" (à la ligne 25).

Une telle optimisation permet d'éviter un à deux accès en mémoire pour chaque champ constant propagé. Puisque la plus grande partie du surcoût de gestion se retrouve dans l'initialisation du descripteur, cette optimisation n'est pas à négliger. Le seul champ constant que l'on ne peut évidemment pas omettre du descripteur est celui de l'adresse de la fonction "proxy" : il demeure l'unique point d'entrée pour les autres processeurs.

La plus grande source d'optimisation se présente cependant lorsque l'on prend avantage d'une convention de passage des paramètres qui emploie la pile d'exécution comme sur l'architecture Intel par exemple [12]. Avec la première approche, chacun des champs du descripteur doit être initialisé, puis possiblement poussé plus tard sur la pile lors de la préparation de l'appel de fonction. Ne peut-on pas faire d'une pierre deux coups et initialiser le descripteur tout en préparant l'appel de fonction ? On peut répondre affirmativement grâce à la stratégie suivante : il s'agit de faire

correspondre le contexte d'appel de la fonction avec une partie du descripteur. Ainsi, il ne suffit que d'ajouter un ou deux champs supplémentaires après avoir poussé les arguments sur la pile. Ces champs serviront au résultat de la tâche s'il existe, ainsi qu'à l'adresse de la fonction "proxy". La figure 3.6 illustre le contenu de la pile d'exécution avant et après l'initialisation du descripteur. Dans un premier temps, nous voyons la pile vide. Ensuite, les arguments d'un appel de fonction y sont empilés. Si nous devions faire l'appel immédiatement, nous n'aurions qu'à brancher à l'adresse de la fonction désirée. Lorsque l'on traite un appel parallèle, il faut aussi y empiler l'adresse de la fonction "proxy" et laisser un espace suffisant pour le résultat de l'appel s'il devait être volé. C'est ce qu'illustre la troisième pile de la figure 3.6.

Cette technique permet de réduire le surcoût d'initialisation du descripteur à deux ou trois instructions machine de plus qu'un appel de fonction standard. La première instruction supplémentaire consiste à faire déplacer le pointeur de pile, alors que les deux autres sont, respectivement, l'affectation de l'adresse de la fonction appelée et celle de la fonction "proxy". Notons cependant que que si la tâche n'est pas volée, il existe aussi un surcoût d'une instruction au moment d'effectuer l'appel afin de replacer le pointeur de pile à l'endroit approprié.

Cette approche apporte une amélioration considérable des performances. En effet, le surcoût de l'initialisation d'une tâche se réduit à deux ou trois instructions seulement. Lorsque l'on compare cette approche à une autre plus portable, nous observons un gain de performance important surtout lorsque les fonctions sont d'une granularité fine.

Cette technique présente toutefois quelques inconvénients non négligeables : elle n'est pas portable sur toutes les plateformes et nécessite l'option de compilation `-fno-defer-pop`. De plus, elle ne peut bénéficier de la propagation des constantes lors de l'initialisation du descripteur. Notons aussi que notre implantation requiert, pour l'instant, que le résultat de l'appel de fonction puisse être retourné via un registre. Néanmoins, les performances amenées par cette technique sont suffisantes pour justifier un compilateur spécifique à la plateforme Intel.

De plus, cette optimisation demande aussi de pouvoir manipuler directement le

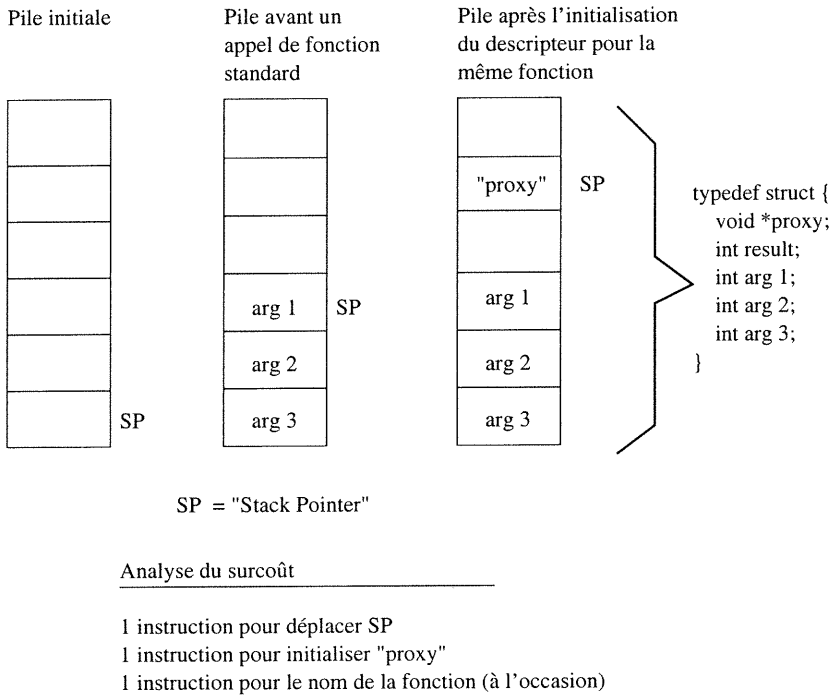


FIG. 3.6 – L'état de la pile sur Intel après l'initialisation d'un descripteur

```
1  register void* _cparSP __asm__ ("esp") ;
2
3  long fib(int x) {
4
5    long f1, f2;
6
7    if ( x < 2 )
8      return 1;
9
10 _cparMoveSP(-12) ;
11 ((_cparRpc0*)_cparSP)->_cparArg1 = x-1 ;
12 ((_cparRpc0*)_cparSP)->_cparProxy = (void*) _cparProxy0 ;
13 _cparPush(SP) ;
14                 { f2 = fib(x-2) ; }
15   f1 = _cparPop(SP,
16          ({ int r=((_cparRpc0*)SP)->_cparResult ;
17             _cparMoveSP(12) ;
18             r ; }),
19          ({ int r ;
20             _cparMoveSP(8) ;
21             r=((int(*)(void))fib)() ;
22             _cparMoveSP(4) ;
23             r ; })) ;
24
25   return f1 + f2;
26 }
```

FIG. 3.7 – Code optimisé pour la plateforme Intel (Fibonacci)

pointeur de pile (`esp` sur Intel). À ce titre, *gcc* offre la fonction `alloca` qui permet d'allouer dynamiquement de l'espace sur la pile. Cependant, cette fonction ne permet pas de désallouer l'espace et désactive une optimisation importante du compilateur : `-fomit-frame-pointer`. Pour remédier à ces problèmes, il était nécessaire d'écrire une extension au compilateur *gcc*. Cette extension prend la forme d'une fonction "built-in" du nom de `_cparMoveSP`. Cette fonction permet simplement de modifier la valeur du registre `esp` en lui additionnant la valeur du paramètre. Étant donné que le compilateur connaît le déplacement du registre, il demeure en mesure de retrouver l'emplacement des autres variables sur la pile. L'extension est facilement appliquée aux sources de *gcc* à l'aide du programme utilitaire `patch`.

3.9 L'empilement d'une tâche

Une fois la tâche allouée et initialisée, il faut la partager avec les autres processeurs en l'empilant sur la pile de tâches du processeur. Cette opération doit être effectuée après chaque initialisation. Cela signifie qu'il s'agit aussi d'une activité à très haute fréquence qui doit être aussi efficace que possible.

Puisque seul le processeur peut empiler des tâches dans sa pile, il n'a aucun besoin de se synchroniser avec les autres. De plus, en conservant un pointeur sur le dessus de la pile dans un registre, nous réduisons au minimum le nombre d'instructions nécessaires à l'empilement. En effet, comme l'illustre la figure 3.8, le code C pour l'empilement d'un descripteur consiste en une assignation en mémoire et une incrémentation du pointeur qui se trouve dans un registre. Cette expression C se traduit habituellement en deux instructions assembleur de type RISC si l'on suppose que l'adresse du descripteur se trouve déjà dans un registre. Sur certaines architectures, notamment celles qui supportent les post et pré-incrémentations, elle peut même se traduire en une seule instruction machine. Avec un surcoût de deux instructions, l'empilement des tâches est une opération relativement peu coûteuse.

```
1    /* Expansion de _cparPush */
2    #define _cparPush(DESC)\
3      _cparTail++[1] = (_cparRPC*)(& DESC)
```

FIG. 3.8 – Macro expansion de _cparPush

```
1    /* Expansion de _cparPop */
2    #define _cparPop(DESC, RESULT, CALL)\
3      ( _cpar_swap_ptr(_cparTop--, NULL),\
4        ((_cparTop<_cparTld->_cparHead) && !_cparRaceForTask(DESC))?\
5          RESULT : CALL);
```

FIG. 3.9 – La macro expansion initiale de _cparPop

3.10 Le dépilement et l'exécution d'une tâche

La réciproque de l'opération d'empilement est évidemment le dépilement de la tâche. Après avoir exécuté le corps de l'expression parallèle, le processeur doit maintenant s'acquitter de l'appel de fonction. Cependant, il se peut que ce dernier ait été volé. Dans ce cas, il devra attendre la fin de son exécution et en profitera pour tenter, à son tour, de voler une tâche à un autre processeur. Le processeur volera une à une les tâches des autres processeurs jusqu'à ce que l'appel soit complété. À ce moment, il récupère le résultat dans le descripteur et poursuit le calcul.

Le code de la figure 3.9 illustre une implantation de la macro `_cparPop`. Lorsqu'on le compare à l'algorithme présenté à la figure 3.3 (à la page 45), on remarque que le premier est une implantation directe de l'autre. Ainsi, le dépilement consiste simplement à retirer la tâche de la pile, ce qui ne nécessite que deux instructions : une assignation en mémoire et une décrémentation du pointeur "tail" qui se trouve dans un registre.

Une fois la tâche dépilée, il faut maintenant déterminer si elle est toujours en suspens, en cours d'exécution, ou si elle est déjà terminée. Dans le premier cas, le processeur pourra effectuer l'appel lui-même et bénéficier de l'optimisation qui consiste à "court-circuiter" la fonction "proxy". Dans le second, il faut attendre que l'autre processeur ait terminé le calcul avant de procéder. Dans le dernier, le processeur peut immédiatement continuer avec le calcul en cours. Ainsi, le test `_cparTail < _cparSelf->Head` permet de déterminer si une tâche a été volée ou non. Lorsque le test est faux, la tâche

```
1  int _cparRaceForTask(_cparRPC *descr) {
2
3    int  is_stolen;
4
5    cpar_lock(&_cparSelf->Lock);
6    is_stolen = ( _cparTail < _cparSelf->Head );
7    _cparSelf->Head = _cparTail;
8    cpar_unlock(&_cparSelf->Lock);
9
10   if (( is_stolen ) && *descr)
11     _cparWaitAndSteal(_cparDesc);
12
13   return (! is_stolen);
14 }
```

FIG. 3.10 – L'implantation de _cparRaceForTask

demeure en suspens et le processeur l'exécute. Dans le cas contraire, une condition de course devient possible et le processeur fait appel à la procédure `_cparRaceForTask` pour la résoudre. Lorsque le contrôle lui revient, il ne reste plus que deux possibilités : la tâche n'avait pas été volée et il doit donc s'en acquitter ou bien son résultat est disponible et il peut procéder avec le reste du calcul.

Le code de la fonction `_cparRaceForTask` se trouve à la figure 3.10. Les lignes 5, 6 et 8 effectuent un "lock" sur le pointeur "tail" afin de déterminer si la tâche a effectivement été volée ou non. Si elle se trouvait sur la pile, le processeur retourne le contrôle au processeur qui l'exécutera directement (sans passer par la fonction "proxy"). Autrement, la tâche a été volée et se trouve soit en cours d'exécution, soit déjà complétée. Le test à la ligne 10 détermine lequel des deux cas s'applique. S'il indique "vrai", la tâche est toujours en cours et le processeur cherche du travail ailleurs. Si, au contraire, il indique "faux", la procédure quitte immédiatement afin de poursuivre le calcul en cours. Notons que l'affectation à la ligne 7 sert uniquement à éviter que le pointeur "head" ne dépasse le pointeur "tail" qui doit agir à titre de limite supérieure.

Le surcoût engendré par toutes ces opérations est relativement bas à l'exception de l'instruction générée par `_cpar_swap_ptr` à la ligne 2 de la figure 3.9. En effet, celle-ci utilise une technique d'écriture de type "write-through" pour propager la valeur jusqu'à la mémoire centrale. Le coût d'une telle écriture se situe entre un et trois

ordres de grandeur de plus qu'une écriture ordinaire de type "copy-back". N'y a-t-il pas un moyen d'éviter ces écritures de type "write-through" ? Après l'examen du vol de tâches, nous verrons qu'il est possible d'éliminer cette instruction "write-through" en la remplaçant par une instruction d'affectation ordinaire de type "copy-back".

3.11 Le vol d'une tâche

Afin de permettre un meilleur emploi des ressources de calcul, il serait opportun de ne pas laisser les processeurs inactifs lorsqu'ils attendent le résultat d'un appel parallèle. Le mécanisme de vol de tâches permet aux processeurs d'aller chercher eux-mêmes du travail dans les piles de tâches des autres processeurs. L'une des conséquences du vol de tâches consiste à offrir un balancement automatique de la charge de travail. En effet, en laissant les processeurs inactifs effectuer le travail à la place de ceux qui sont surchargés, on permet une pleine utilisation des processeurs.

Cette technique possède cependant un coût relativement élevé. Elle doit donc être employée au minimum. De plus, une recherche active lorsqu'il n'y a pas de travail disponible peut engendrer un gaspillage des ressources de la machine, ce qui est évidemment indésirable. Deux techniques sont employées pour réduire l'impact du vol de tâches sur le système. La première cherche à minimiser la fréquence des vols en prenant un maximum de travail à la fois aux autres processeurs. La seconde joue sur les priorités d'exécution des processeurs afin d'éviter de gaspiller les ressources inutilement.

3.11.1 Le mécanisme de vol

Un processeur peut accéder à la pile des autres processeurs grâce à une liste globale mise à sa disposition. En employant le protocole d'exclusion mutuelle, un processeur peut accéder à la pile d'un autre processeur. Par une simple indirection du pointeur "head" d'un autre processeur, il peut déterminer si cette pile contient du travail ou non. Lorsque l'indirection est nulle, aucun travail n'est disponible. Dans le cas contraire, cela indique qu'au moins une tâche s'avère potentiellement disponible et

```
1  _cparRPC* _cparTrySteal(void) {
2    ...
3    if ( victim->Head[1] != NULL ) {
4
5      cpar_lock( &victim->Lock ) ;
6      victim->Head++ ;
7      _cpar_fence() ;
8
9      descr = *victim->Head ;
10
11     if ( descr == NULL )
12       victim->Head-- ;
13
14     cpar_unlock( &victim->Lock ) ;
15   }
16
17   return descr ;
18 }
```

FIG. 3.11 – Code pour le vol d'une tâche

il tente de l'obtenir. S'il l'obtient, il exécute la tâche par l'entremise de la fonction "proxy". Si, au contraire, il n'a pas pu l'obtenir, il passe alors à la pile du prochain processeur.

Lorsqu'un processeur termine une tâche qu'il a volée, il doit mettre l'adresse de la fonction "proxy" à zéro indiquant à la victime que la tâche est complétée et que s'il y a un résultat, il est désormais disponible. De cette façon, la victime peut facilement se synchroniser et poursuivre le calcul en cours.

La figure 3.11 illustre le code du noyau concernant le vol d'une tâche. Ce code est presque identique à celui que l'on retrouve à la figure 3.3 (à la page 45). En effet, mis à part les appels à la fonction `_cpar_fence` on ne pourrait distinguer les deux fonctions.

La fonction `_cpar_fence` consiste simplement à s'assurer que toutes les opérations en mémoire (lectures et écritures) sont complétées avant de procéder. On peut ainsi s'assurer qu'une écriture en mémoire est visible de tous avant de procéder avec le calcul. Notons aussi que les fonctions `cpar_lock` et `cpar_unlock` possèdent cette même caractéristique. Toutes ces fonctions emploient l'instruction générée par `_cpar_swap`. Elles sont donc aussi assez coûteuses en termes de performances. Cependant, leur utilisation dans le cadre d'un vol de tâche s'avère essentiel et relativement peu coû-

teux en comparaison avec l'ensemble du processus.

Ainsi, après avoir identifié une victime, le voleur commence par déterminer s'il y a ou non une tâche à voler dans la pile de la victime. S'il n'en existe pas, la procédure quitte immédiatement. Autrement, il obtient l'exclusivité de la tâche, puis incrémente le pointeur "head" (les lignes 5 et 6). L'appel à `_cpar_fence` qui suit, assure que la nouvelle valeur du pointeur "head" soit visible de la victime avant de continuer. Enfin, il tente d'obtenir la tâche (ligne 9). S'il réussit, il la retourne à la fonction appelante. Autrement, il restitue le pointeur "head" à sa valeur originale (ligne 12). L'appel à `cpar_unlock` nous assure que la valeur restituée du pointeur "head" est à nouveau connue de la victime.

3.11.2 Minimiser la fréquence des vols

Comme le vol de tâches demeure une activité coûteuse, il est important d'en minimiser la fréquence. Ainsi, n'est-ce pas un hasard si la structure de la liste de tâches reproduit celle d'une pile. En effet, les caractéristiques LIFO de la pile de tâches font en sorte que les tâches les plus anciennes se retrouvent en bas. C'est aussi au bas de la pile que l'on retrouve initialement le pointeur "head". En se fondant sur l'hypothèse que des tâches lancées plus tôt dans l'exécution d'un programme (donc au bas de la pile) contiennent probablement plus de travail que celles qui se trouvent sur le dessus de la pile, on peut avancer que le vol d'une tâche à partir du bas de la pile serait plus "rentable" pour un voleur que s'il le faisait à partir du haut [9, 13]. Ainsi, comme Cilk et ParSubC, CPar favorise donc un style de programmation "diviser pour régner".

3.11.3 Priorité d'exécution des processeurs

Un programme est rarement exécuté seul sur un ordinateur. Plusieurs autres programmes sont toujours en exécution simultanément. Cette affirmation devient d'autant plus vraie lorsque l'on se trouve sur une machine multi-utilisateurs. Pour être le plus "CPU Friendly" possible, nous devons éviter l'attente active. Cependant, nous

voulons aussi conserver un temps-réponse rapide lorsque le travail se présente ou lorsque le programme est terminé. L'approche que CPar privilégie consiste à donner une priorité d'exécution inférieure à la normale à tous les processeurs. Ainsi, le programme CPar libère les ressources de calcul lorsqu'il ne peut les employer.

Notons cependant qu'à cause de ce jeu de priorités, il se peut que les accélérations réelles soient inférieures à celles attendues. Cette situation pourrait se présenter lorsqu'un ordinateur est très chargé. La priorité d'exécution des processeurs est un paramètre que l'on peut spécifier à la ligne de commande. Ainsi, il est possible d'effectuer un calcul avec le temps libre de l'ordinateur en lui spécifiant une priorité de 20 avec l'option `--nice 20` à la ligne de commande.

3.12 Le protocole d'exclusion mutuelle relaxé

Comme on le sait, l'utilisation des écritures de type "write-through" possède un coût élevé. Même si l'impact de ces opérations est petit en comparaison avec le processus de vol de tâches, il n'en est pas de même pour le dépilement d'une tâche. Quelles sont les conséquences possibles de l'utilisation d'une écriture ordinaire de type "copy-back" plutôt que la plus coûteuse écriture "write-through" dans l'expansion de la macro `_cparPop`? D'une part, il en résulte un gain considérable de performances. De l'autre, le protocole, tel que défini, n'est plus en mesure de garantir l'exclusivité d'accès à une tâche. En d'autres termes le programme n'est plus valide car une tâche pourrait être exécutée plus d'une fois.

La figure 3.12 illustre un scénario possible lors de l'exécution d'un programme CPar qui emploie des écritures de type "copy-back". Initialement, il ne reste plus qu'une tâche sur la pile de la victime. La victime la retire alors de la pile (étape A). Cependant, parce que l'affectation à `NULL` est de type "copy-back", sa nouvelle valeur met un certain temps à être visible au reste du système. Ce délai est représenté par une barre verticale grise. Ainsi, lorsqu'un voleur veut aussi retirer la tâche de la pile (étape B), rien ne l'en empêche car, de son point de vue, la tâche s'y trouve toujours. Il la retire donc de la pile et en incrémentant le pointeur `victim->Head`. Cependant,

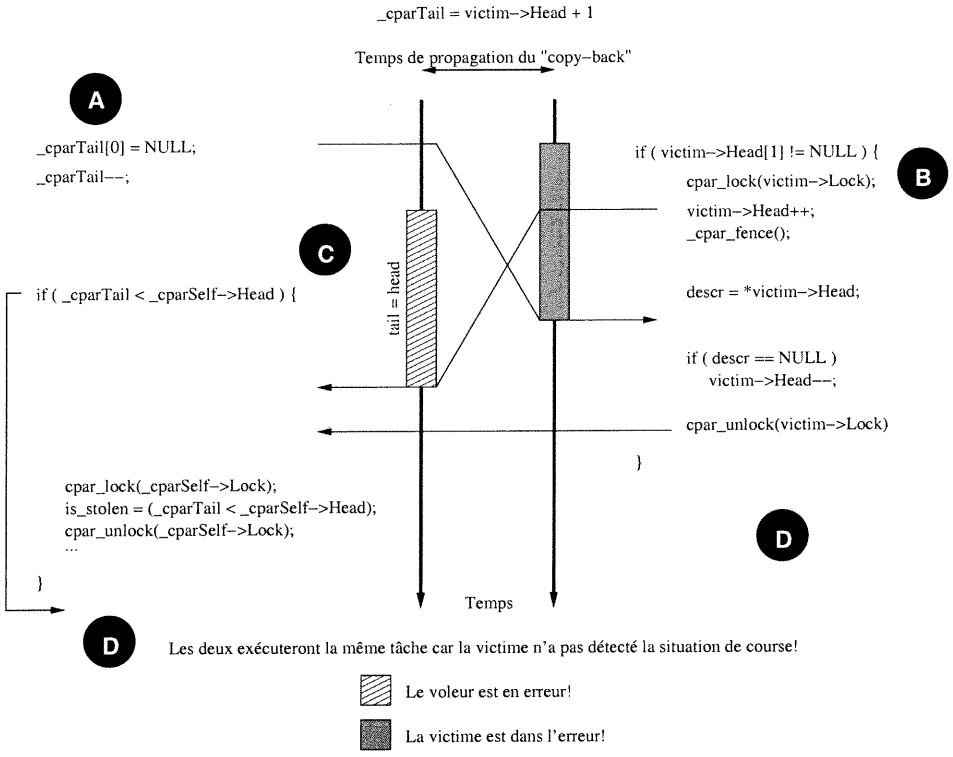


FIG. 3.12 – Conséquence d'une écriture "copy-back" sur le protocole

```
1   /* Ancienne expansion de _cparPop */
2   #define _cparPop(DESC, RESULT, CALL)\
3     ( _cpar_swap_ptr(_cparTop--, NULL),\
4       ((_cparTop<_cparTld->_cparHead) && !_cparRaceForTask(DESC)) ?
5         RESULT : CALL) ;
6
7   /* Nouvelle expansion de _cparPop */
8   #undef _cparPop
9   #define _cparPop(DESC, RESULT, CALL)
10    ( *_cparTop-- = NULL,
11      ((_cparTop<=_cparTld->_cparHead)&& !_cparRaceForTask(DESC))
12        RESULT : CALL) ;
```

FIG. 3.13 – La macro expansion finale de _cparPop

comme l'incrémentation met aussi un certain temps à se propager dans le système (la barre verticale hachurée), la victime n'est plus en mesure de détecter la situation de course qui a lieu à l'étape C. Par conséquent, le voleur et la victime exécuteront la tâche à l'étape D.

En d'autres termes, l'utilisation d'une écriture de type "copy-back" ne permet plus à la victime de détecter correctement la situation de course. Lorsque celle-ci effectue le test `_cparTail < _cparSelf->Head`, la nouvelle valeur de `_cparSelf->Head` n'est pas encore disponible. De son point de vue, les deux pointeurs sont toujours égaux. Par conséquent, en employant une écriture de type "copy-back", une situation de course se présente aussi lorsque les pointeurs "head" et "tail" sont égaux. Le protocole d'exclusion mutuelle peut donc être modifié afin de tenir compte de ce nouvel élément. Cependant, modifier la condition qui permet de détecter une situation de course n'est pas suffisant. En effet, si deux processeurs choisissent la même victime en même temps, le même problème se présente à nouveau. Toutefois, si les victimes sont choisies à tour de rôle, par opposition à un choix aléatoire, on s'assure que le nouveau protocole d'exclusion mutuelle fonctionne correctement.

La figure 3.13 illustre la macro expansion finale de `_cparPop`. Pour saisir l'importance de ce gain de vitesse gagné en relaxant le protocole original, le tableau 3.2 illustre les temps d'exécution de quelques programmes en employant les deux approches (ces programmes, de même que les plateformes utilisées pour leur exécution, sont présentés au chapitre 4). Comme on peut le constater, cette modification peut augmenter considérablement les performances des problèmes de granularité fine.

			"Write-Through"		"Copy-Back"	
Programme	Plateforme	T_s	T_1	$\frac{T_1}{T_s}$	T_1	$\frac{T_1}{T_s}$
fib(36)	SPARC	2.16	3.84	77.8%	2.92	35.2%
scan(1e7)	SPARC	2.77	4.41	59.2%	3.52	27.1%
somme(3e7)	SPARC	3.36	5.23	55.7%	4.22	25.6%
queens(14)	SPARC	4.67	6.43	37.7%	6.25	33.8%
fib(36)	Intel	4.92	8.77	78.3%	4.90	0.0%
scan(4e6)	Intel	2.68	4.01	49.6%	2.84	6.0%
somme(10e7)	Intel	2.90	4.39	51.4%	3.01	3.8%
queens(13)	Intel	2.74	3.45	25.9%	2.85	3.6%

T_s : temps d'exécution du programme séquentiel.
T_1 : temps d'exécution du programme parallèle sur 1 processeur.
$\frac{T_1}{T_s}$: mesure communément appelée "overhead" ou surcoût.

TAB. 3.2 – Comparaison des écritures "copy-back" et "write-through"

3.13 La synchronisation des processeurs

Le vol de tâches rend nécessaire un protocole de synchronisation efficace. Comme on le sait, le modèle de synchronisation implicite de CPar est moins général qu'un modèle de synchronisation explicite. Quel est donc l'intérêt de cette synchronisation implicite par opposition à une synchronisation explicite comme celle de Cilk ? D'autant plus qu'une synchronisation explicite possède l'avantage de généraliser le modèle de programmation. D'abord, la synchronisation implicite se trouve moins sujette à l'erreur humaine que la synchronisation explicite. Cependant, le potentiel d'atteindre des performances plus élevées demeure la principale raison de l'emploi d'une synchronisation implicite. En effet, le modèle de synchronisation de CPar ne nécessite aucun "lock" alors qu'un modèle plus général se doit de les employer. En effet, le modèle de CPar est plus simple et surtout plus rapide à l'exécution que la synchronisation explicite. Deux facteurs sont responsables pour ce gain de vitesse : l'interprétation du mot réservé `par` et l'efficacité du mécanisme de synchronisation.

3.13.1 L'interprétation du mot réservé "par"

En ce qui concerne le mot réservé "par", deux interprétations sont possibles. Dans la première, la tâche courante arrivant au point de parallélisation lance deux tâches

```
1  /* Tache initiatrice */      1 /* Autres taches */
2  while ( counter > 0 )        2 lock(counter_lock) ;
3    do anything                3 counter-- ;
4                               4 unlock(counter_lock) ;
```

FIG. 3.14 – Un "join" entre n-1 tâches et la tâche initiatrice

concurrentes et se bloque jusqu'à ce que les deux aient terminé. Dans la seconde, la tâche courante lance une tâche en parallèle avec elle-même et continue son exécution avec le corps de la construction parallèle jusqu'au point de synchronisation. Ensuite, elle attend que l'appel lancé en parallèle soit terminé.

CPar privilégie la seconde interprétation pour deux raisons. D'abord, rappelons-nous que l'initialisation d'un descripteur reste une opération coûteuse. En ne lançant qu'une seule tâche plutôt que deux, nous n'avons qu'un descripteur à initialiser. Ce facteur entraîne donc un surcoût de gestion moins élevé. Ensuite, il permet une simplification du mécanisme de rendez-vous, qui à son tour, minimisera les pertes de performances.

3.13.2 Les "joins" dans un modèle explicite

Le problème du "join" consiste à synchroniser deux ou plusieurs processeurs en un point fixé et permettre ainsi à plusieurs flux de contrôle de n'en former qu'un seul. Pour effectuer un "join" entre n processeurs, il suffit d'un compteur dont la valeur initiale est de n. Lorsque l'un d'eux arrive au point de rencontre, il décrémente ce compteur. Lorsque celui-ci est à zéro le calcul peut continuer. On peut cependant supposer qu'au moment où un processeur est en mesure de détecter le "join", il s'y trouve déjà et il ne reste plus qu'à effectuer un "join" entre les $n-1$ autres processeurs. Le pseudo-code C à la figure 3.14 illustre comment l'on pourrait effectuer un tel "join". La nécessité de synchroniser l'accès au compteur est une source de sérialisation et une perte d'efficacité.

```
1  /* Tache initiatrice */      1 /* Autres taches */
2  while ( counter > 0 )        2 counter = 0 ;
3    do anything                3
```

Un "join" avec $n-1$ tâches, nécessite $n-1$ exécutions de cette section de code.

FIG. 3.15 – Un "join" entre une tâches et la tâche initiatrice sous CPar

3.13.3 Les "joins" dans le modèle de CPar

Lorsque le "join" se fait entre deux tâches seulement, l'implantation devient triviale. En effet, puisqu'un processeur doit effectuer le "join" avec un seul autre processeur, il n'est plus nécessaire d'obtenir l'exclusivité du compteur car elle devient implicite. Le code de la figure 3.15 illustre la simplicité de l'implantation d'un tel "join".

Afin de comparer cet algorithme au précédent, il faut pouvoir simuler un "join" entre n processeurs. Le "join" entre un processeur et les $n-1$ autres est équivalent à faire $n-1$ "joins" entre ce même processeur et chacun des autres pris individuellement en des points de rencontre consécutifs. La perte de parallélisme n'est qu'illusoire, car elle est passée de la décrémentation du compteur à une séquence de joins. Cependant, l'absence de lock réduit la perte de performance.

En pratique, il nous faut un compteur accessible à la fois du créateur de la tâche et des autres processeurs. Dans un programme CPar, le seul mot mémoire facilement accessible de la victime comme du voleur est le premier mot du descripteur de tâche. Celui-ci correspond à l'adresse de la fonction "proxy". Comme cette dernière n'est utile que pour reconstruire l'appel de fonction, nous pouvons aussi l'utiliser comme variable commune après ce même appel. À cette fin, on peut convenir qu'au moment où l'adresse de la fonction "proxy" est à zéro, le résultat de l'appel parallèle, s'il existe, est disponible. Toutefois, le surcoût économisé par l'élimination des "lock" n'est pas très important. En effet, la synchronisation a lieu à l'intérieur du processus de vol de tâches qui s'avère beaucoup plus coûteux. Les gains de vitesse proviennent plutôt d'une réduction du nombre de tâches générées.

3.14 Synthèse et exemple détaillé

Nous proposons un ensemble de techniques et de stratégies pour rendre le compilateur et son noyau aussi efficace que possible. Parmi celles-ci on retrouve :

- La création paresseuse des tâches et le vol de tâches qui permettent la création d'un très grand nombre de tâches ainsi que le balancement automatique de la charge.
- L'accès à la pile locale d'un processeur par le biais d'un registre qui améliore les performances globales du système.
- Une structure de données flexible pour les descripteurs de tâches qui permet de prendre avantage de certaines conventions de passage des paramètres.
- L'emploi des fonctions "proxy" pour simplifier la reconstruction d'un appel parallèle ainsi que la synchronisation des processeurs.
- L'utilisation des écritures de type "copy-back" plutôt que "write-through" pour améliorer considérablement les performances du dépilement d'une tâche.
- Un modèle de synchronisation simple et efficace.

Même si ces techniques n'ont pas toutes le même impact, elles contribuent néanmoins aux performances générales de CPar et de son noyau. Il est cependant intéressant de comparer le code généré d'un programme parallèle à celui d'un programme séquentiel.

À ce titre, la figure 3.16 illustre le code optimisé tel que généré pour la plateforme Intel. Les instructions encadrées sont celles ajoutées par le compilateur CPar. Comme on peut le constater, dans la majorité des cas, un appel parallèle n'engendre qu'un surcoût total de 12 instructions dont seulement cinq font référence en mémoire. Ainsi, dans cet exemple, l'initialisation du descripteur de tâche engendre deux instructions de plus qu'un appel de fonction ordinaire. Il nous faut aussi deux instructions pour empiler la tâche. Plus tard, deux instructions sont nécessaires pour la retirer de la pile. Il en faut à ce moment 5 autres pour détecter s'il y a ou non une situation de course. Lorsqu'il n'y a pas eu de vol de tâche, il faut ajuster le pointeur de la pile d'exécution pour que cette dernière reflète un appel de fonction ordinaire. Cette opération nous coûte une instruction pour un total de 12 instructions.

On peut aussi remarquer, sur cet exemple particulier, que le code sur les feuilles de l'arbre d'appels parallèles semble plus efficace que les mêmes appels séquentiels. Ce type d'optimisation dépend cependant du compilateur *gcc* et non pas de CPar.

Après cette présentation des techniques et stratégies employées par le compilateur CPar pour réduire le surcoût engendré par l'utilisation du parallélisme, il convient de les voir à l'œuvre. Le prochain chapitre propose donc une évaluation empirique du compilateur.

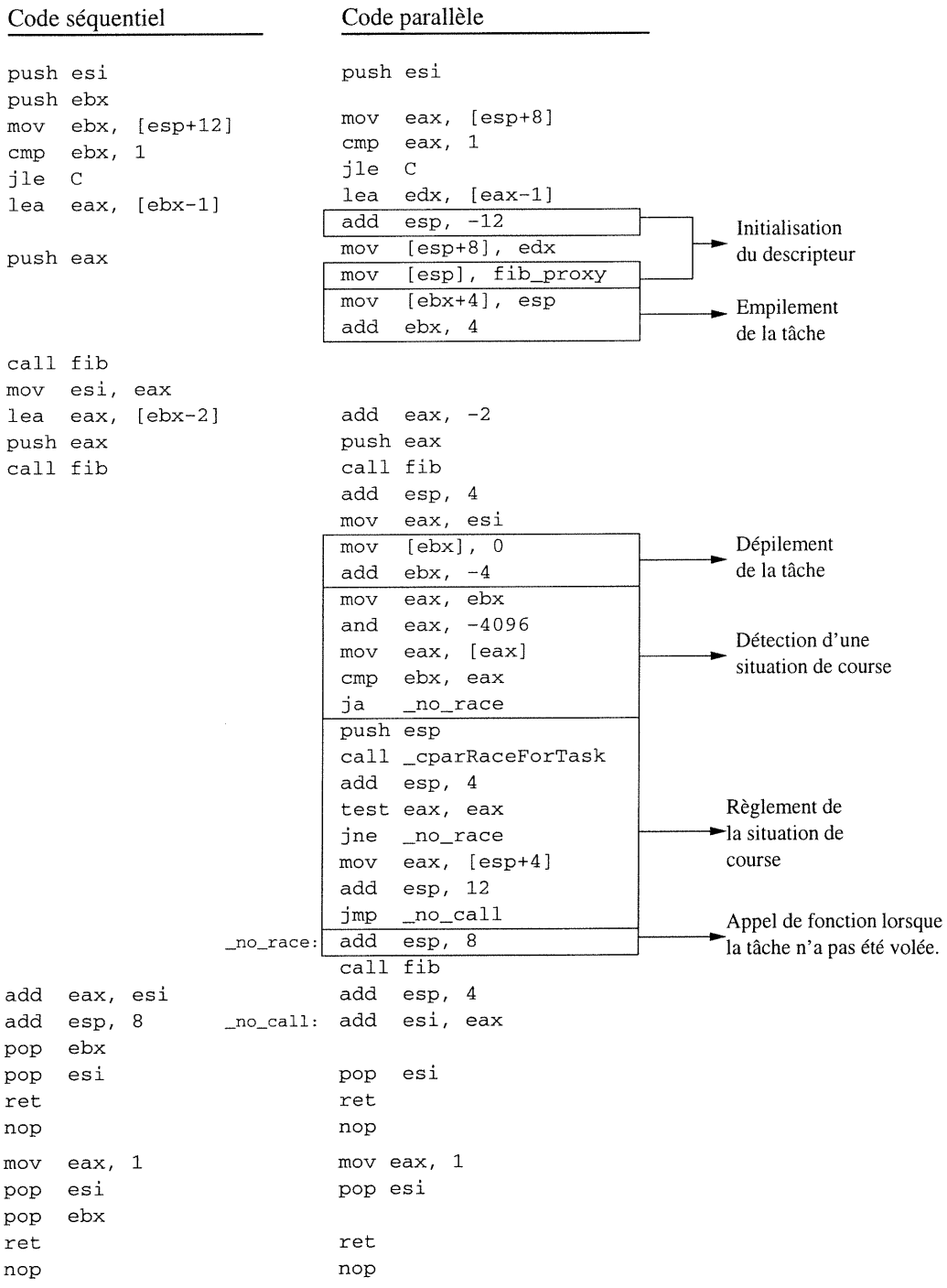


FIG. 3.16 – Génération du code optimisé sur Intel

Chapitre 4

Évaluation

4.1 L'évaluation du système

L'objectif de CPar consiste à procurer au programmeur un moyen efficace et peu coûteux d'exprimer le parallélisme de contrôle. Notre compilateur vise ainsi à minimiser le surcoût engendré par toutes les activités de gestion des tâches. Afin d'atteindre cet objectif, nous avons employé les meilleures caractéristiques de ParSubC et en avons amélioré les performances dans le contexte d'un système parallèle à mémoire partagée. Les techniques utilisées, ainsi que leurs implantations, promettent des résultats intéressants qu'il nous faut cependant vérifier expérimentalement. Nous allons donc comparer les performances respectives de ces deux compilateurs. Ensuite, nous comparerons les performances des compilateurs CPar et Cilk. L'usage de Cilk s'est répandu grâce à l'efficacité des programmes que son compilateur génère pour des applications d'une granularité fine. Il sera intéressant de voir comment notre implantation de CPar peut se comparer à celle de Cilk. Afin de mieux évaluer le comportement des programmes générés par le compilateur CPar, une batterie de tests représentatifs s'avère nécessaire. Sont inclus dans cet ensemble de tests des programmes variés de calculs numériques, de recherche arborescente, de calculs matriciels et de recherche opérationnelle.

Certains programmes sont simples et servent à mesurer les performances de CPar

pour des calculs parallèles d'une granularité fine. D'autres sont conçus pour montrer les forces de CPar lorsqu'il est déployé à grande échelle. Ces programmes sont les suivants : *fib, somme, scan, queens, knap, poly, abi, mmul* et *tsp*. Ces programmes sont disponibles sur Internet à l'adresse http ://www.iro.umontreal.ca/~cpar/.

La fonction de Fibonacci (*fib*) : La fonction de Fibonacci possède la particularité d'être simple, mais néanmoins représentative d'une fonction dont la granularité est fine. Le programme *fib* calcule cette fonction avec un algorithme récursif naïf. Il existe évidemment des algorithmes plus efficaces pour calculer cette suite. Cependant, celui-ci permet de mettre en évidence le surcoût de gestion que CPar peut engendrer dans des conditions de "pire cas", car il est difficile de créer une fonction récursive qui requiert moins de temps de calcul et qui soit d'une granularité plus fine.

La sommation des entrées d'un vecteur (*somme*) : Le programme somme calcule la *somme* de toutes les entrées d'un vecteur X de taille n. Ce dernier présente une granularité comparable à celle de *fib*, mais nécessite toutefois le parcours d'une plus grande plage mémoire.

Le préfixe parallèle (*scan*) : Le programme *scan* calcule le préfixe parallèle d'un vecteur X de taille n. Chaque élément du vecteur est remplacé par la somme de lui-même et de toutes les entrées du vecteur le précédant, plus précisément : $X_i = \sum_{j=1}^{i} X_j$.

Le problème des reines (*queens*) : Le programme *queens* calcule le nombre de solutions au problème des n reines. Une solution à ce problème consiste à placer n reines sur un échiquier $n \times n$ de telle sorte qu'aucune des reines ne puisse en attaquer une autre. La fonction récursive présente la particularité de posséder un grand nombre d'arguments, ce qui devrait mettre en évidence les surcoûts reliés à l'initialisation des descripteurs. Ce programme est tiré de la liste des exemples dans les sources de ParSubC.

Le problème du sac alpin (*knap*) : Le programme *knap* employé ici est celui du sac alpin binaire tiré de la liste des exemples de la distribution de Cilk 5.2. Il consiste à maximiser la fonction $f(x_1, x_2, ..., x_n) = \sum_{i=1}^{n} x_i p_i$ avec la contrainte que $\sum_{i=1}^{n} x_i w_i \leq W$ sachant que $x_i \in \{0, 1\}$, p_i, w_i et W sont des entiers positifs. L'algorithme effectue une recherche de type "branch and bound" afin de trouver une solution optimale à un problème donné. La version parallèle lance une recherche en parallèle sur toutes les branches de l'arbre de recherche.

Le calcul d'un polynôme (*poly*) : Le programme *poly* calcule le carré d'un polynôme de n termes dont les coefficients sont entiers et l'évalue à $x = 1$. Le polynôme résultant n'est pas conservé.

Le problème du tri (*abisort*) : Le programme *abisort* effectue un tri sur n termes. L'algorithme employé est de type bitonique et il provient de la batterie de tests fournie avec ParSubC. Ce programme contient toutefois une bonne proportion de code séquentiel où le parallélisme est impossible. Sur un petit nombre de processeurs, l'impact devrait être négligeable. Sur un grand nombre de processeurs, ce n'est pas le cas. En revanche, plus la taille du problème est grande moins important sera l'impact du code séquentiel.

La multiplication de matrices (*mmul*) : Le programme *mmul* effectue la multiplication de deux matrices avec un algorithme de l'ordre $O(n^3)$. Ce programme sert à souligner l'efficacité de CPar en comparant une version entièrement séquentielle et itérative de la multiplication de matrices à une version parallèle qui fait appel à la récursivité pour effectuer la division du travail. Notons cependant que la plus petite tâche consiste à calculer le produit scalaire de deux vecteurs.

Le problème du commis voyageur (*tsp*) : Le programme *tsp* propose une méthode heuristique pour résoudre le problème du voyageur de commerce (en anglais "traveling salesman problem") qui consiste à trouver le chemin de longueur minimum passant une seule fois par tous les points d'un ensemble de points et revenant au

point d'origine. Le programme CPar proposé s'inspire des idées énoncées dans Reeves [14]. Essentiellement, le programme utilise un algorithme génétique qui permet à une population de solutions candidates plus ou moins valables d'évoluer vers une solution qui s'approche de plus en plus de l'optimale. L'algorithme procède par itération successive que l'on nomme "génération". Chaque solution candidate de la génération $i + 1$ est construite à partir de deux solutions de la génération précédente. Ces solutions sont choisies de telle sorte que les meilleures aient plus de chances d'être sélectionnées. L'idée est d'obtenir la meilleure solution possible tout en minimisant le nombre de générations et donc, le temps de calcul. Alors que le programme séquentiel construit une à une les solutions candidates de la prochaine génération, le programme parallèle les construit toutes en parallèle. Soulignons que les instances employées sont tirées de la distribution de TSPLIB [15].

4.2 Plateformes de tests

Les tests sont effectués sur les architectures Intel et SPARC sur les plateformes respectives Linux et Solaris. Une description sommaire de chacune de ces machines permet de mieux situer le contexte dans lequel nous avons procédé à l'évaluation de CPar. L'ordinateur Intel possède quatre processeurs physiques Pentium à une cadence de 150 MHz chacun. La mémoire centrale est de 128Mo avec un espace temporaire de 512Mo. L'ordinateur SPARC est un Sun Enterprise 10000 avec 64 processeurs SPARC à une cadence de 400 MHz chacun. Sa mémoire centrale est de 64Go et l'espace temporaire est de 128Go.

4.3 La compilation des programmes tests

Afin d'obtenir des résultats réalistes, les meilleures options d'optimisation des compilateurs ont été choisies. Dans le cas des programmes séquentiels, ces options de compilation sont :

```
gcc -O2 -fomit-frame-pointer -D__CPAR_SERIAL__
```

En revanche, les options de compilation pour les programmes parallèles sont légèrement différentes :

```
gcc -O2 -fomit-frame-pointer [-fno-defer-pop]
```

La macro `__CPAR_SERIAL__` est parfois employée par le programme séquentiel pour optimiser ses performances. De plus, sur la plateforme Intel, il est nécessaire d'utiliser l'option de compilation `-fno-defer-pop` lorsqu'on utilise l'optimisation CPar propre à cette plateforme (voir la section 3.8).

4.4 Définition des mesures

Les tests tentent de mesurer les performances d'un programme CPar. Celles qui nous intéressent particulièrement sont : le surcoût de gestion, le surcoût de communication, l'accélération, ainsi que l'efficacité. Lors de l'analyse des performances de nos programmes parallèles, nous nous penchons sur la notion d'exécution. Celle-ci se définit comme étant la résolution d'une instance particulière d'un problème par un programme parallèle. Une exécution est caractérisée par : le nombre de processeurs p utilisés lors de l'exécution, ainsi que par ses paramètres d'entrées en particulier, la taille de l'instance du problème. Ainsi, on peut effectuer plusieurs mesures sur l'exécution d'un programme [5] :

Les temps d'exécution $T(p)$ et T_s : La mesure la plus simple consiste à mesurer le temps total réel écoulé lors de l'exécution du programme sur p processeurs. Nous notons $T(p)$ le temps d'exécution d'un programme parallèle sur p processeurs. De plus, notons T_s la mesure du temps d'exécution du programme séquentiel équivalent : c'est-à-dire en remplaçant le mot réservé "par" remplacé par un " ;". Notons que le temps mesuré correspond au temps réel écoulé calculé à l'aide de la fonction système `gettimeofday`.

Le surcoût de gestion $B(p)$: Il existe habituellement un coût associé à l'exploitation du parallélisme. La synchronisation et la communication n'en sont que

deux causes possibles. Il existe cependant un moyen de capturer dans une seule quantité la totalité du travail supplémentaire effectué par le programme parallèle : $B(p) = \frac{p \times T(p) - T_s}{T_s}$. Cette mesure, normalisée par rapport au temps d'exécution séquentiel, nous indique d'une manière relative, le surcoût qu'engendre l'utilisation du parallélisme. Notons que cette mesure est intéressante lorsque $p = 1$ car elle représente alors le surcoût minimum que doit engendrer l'exécution du programme parallèle même s'il n'y a aucun parallélisme possible (il n'y a qu'un seul processeur). Soulignons que cette mesure, lorsque $p = 1$, est surtout connue sous le nom anglais d'"overhead".

Le surcoût de communication $B_c(p)$: La communication est un facteur limitatif important dans les programmes parallèles. Il s'avère donc intéressant de mesurer le surcoût attribuable à la communication entre processeurs. Comme l'exécution du programme parallèle sur un seul processeur ne présente aucun surcoût relié à la communication, il est possible d'employer cette mesure afin d'évaluer le surcoût de gestion attribuable à la communication. Le surcoût de communication se calcule ainsi : $B_c(p) = \frac{p \times T(p) - T(1)}{T_s}$. Cette mesure normalisée nous donne une approximation du temps d'exécution consacré à l'exécution de code séquentiel.

Le surcoût d'initialisation : Tous les programmes CPar possèdent une section de code séquentiel qui consiste à lancer l'exécution des processeurs logiques. Leur création possède un impact négligeable lorsque la taille des instances est grande, mais peut être plus important lorsque le temps d'exécution est court par rapport au nombre de processeurs. Nous nommons surcoût d'initialisation cette composante du surcoût de communication.

L'accélération $S(p)$: L'accélération est une des mesures intéressantes sur l'exécution d'un programme parallèle. Elle se calcule par le rapport entre le temps d'exécution séquentiel et le temps d'exécution parallèle : $S(p) = \frac{T_s}{T(p)}$. Une accélération idéale a une valeur de p, indépendamment de la taille du programme. Cependant, selon la loi de Amdahl [13], l'accélération maximum d'un programme est limitée par

les calculs séquentiels dans ce dernier.

L'efficacité $E(p)$: La mesure de l'efficacité est une normalisation de l'accélération. L'efficacité est le rapport entre l'accélération et le nombre de processeurs, $E(p) = \frac{S(p)}{p}$. Ainsi, une efficacité de 1 signifie que le programme parallèle utilise chaque processeur au même rendement que le programme séquentiel équivalent (évidemment la performance du programme sera plus élevée). Une efficacité proche de 1 signifie que le programme perd peu de temps à la gestion des aspects parallèles du programme.

4.5 Tests sur la plateforme SPARC

Le Sun Enterprise 10000 offre une plateforme qui permet de mesurer les performances de CPar sur l'architecture SPARC. De plus, le nombre important de processeurs permet de mieux mesurer l'effet du surcoût de communication sur l'exécution des programmes. Puisqu'il est difficile d'obtenir l'exclusivité des 64 processeurs de la machine, nous nous sommes limité à l'utilisation de 58 d'entre eux. Comme certaines exécutions restent relativement longues, il est préférable d'effectuer une moyenne des temps d'exécution sur plusieurs exécutions afin de réduire l'impact des influences externes (par exemple, l'exécution d'autres programmes) sur nos mesures. Chacune des mesures effectuées consiste donc en une moyenne des temps de six exécutions. Les résultats de nos mesures se retrouvent dans le tableau 4.1. Le programme *tsp* ne s'y trouve pas parce qu'il fait l'objet d'une prochaine section.

Notons d'abord que les performances générales semblent satisfaisantes. Hormis le programme *abisort*, l'accélération des programmes varie entre 39.9 et 57.5. Le plus important "overhead" ($B(1)$) mesuré est de 40.6%. Il n'est pas surprenant de constater que c'est le programme *fib* qui affiche ce surcoût car il possède la plus fine granularité de tous les programmes mesurés. De plus, on remarque que notre implantation parallèle et récursive de la multiplication de matrices est parfois plus rapide que la version séquentielle qui utilise trois boucles imbriquées. Cette constatation est assez

surprenante sachant qu'il est normalement plus efficace d'utiliser des boucles imbriquées que des appels récursifs de fonctions. Nous croyons cependant que ce gain de performance est causé par un meilleur parcours des plages mémoire et n'est donc pas représentatif des performances générales de CPar.

Soulignons ensuite que le surcoût de communication est relativement faible lorsque les instances des programmes sont de grande taille. Comme on peut s'y attendre, cette mesure reflète de moins bon résultats lorsque ces instances sont de plus petite taille car le surcoût de communication est alors dominé par le surcoût d'initialisation du programme (le temps requis pour lancer les 58 processus légers). Il est intéressant néanmoins de constater que le surcoût de communication reste faible même en présence d'un nombre important de processeurs.

Enfin, on remarque que le surcoût d'initialisation des processeurs et l'"overhead" sont les principales causes de la dégradation des performances par rapport au programme séquentiel. Cependant, le surcoût d'initialisation des processeurs ne possède un impact significatif que si les instances sont de petite taille en comparaison avec le nombre de processeurs. Le programme *mmul* est un exemple qui illustre bien l'impact du surcoût d'initialisation. Lorsque l'instance est de petite taille (1024), l'efficacité du programme parallèle n'est que de 92.2% et croît pour atteindre 99.1% lorsque la taille du problème augmente à 2048.

Les tests sur l'ordinateur Sun Enterprise 10000 mettent à l'épreuve la stratégie de compilation portable. Bien que les résultats préliminaires semblent bons, la situation pourrait s'améliorer en prenant avantage de l'architecture sous-jacente, ce que nous avons tenté d'exploiter sur la plateforme Intel.

4.6 Tests sur la plateforme Intel

L'ordinateur Intel sur lequel nos tests sont effectués ne possède que quatre processeurs. Dans ce contexte, l'étude des problèmes de taille importante n'apporte aucune contribution. Il s'avère aussi moins intéressant de discuter de surcoût de communication à cause du nombre restreint de processeurs. Par conséquent, nous

Programme	T_s	$T(1)$	$T(58)$	"Overhead" $B(1)$	Surcoût comm. $B_c(58)$	Accél. $S(58)$	Efficacité $E(58)$
fib(42)	38.30	51.00	0.95	33.2%	10.7%	40.3	69.5%
fib(44)	101.03	131.95	2.36	30.6%	4.9%	42.8	73.8%
fib(46)	243.12	341.87	6.09	40.6%	4.7%	39.9	68.8%
somme(2e8)	24.57	28.59	0.52	16.4%	5.5%	47.3	81.6%
somme(6e8)	73.06	87.76	1.55	20.1%	6.4%	47.1	81.2%
somme(1e9)	116.17	140.23	2.46	20.7%	2.1%	47.2	81.4%
queens(15)	33.04	39.58	0.78	19.8%	17.1%	42.4	73.1%
queens(16)	220.75	267.69	4.74	21.3%	3.3%	46.6	80.3%
queens(17)	1588.18	1905.12	33.20	22.3%	1.3%	47.8	82.4%
scan(2e8)	56.20	69.00	1.28	22.8%	9.3%	43.9	75.7%
scan(4e8)	112.23	138.91	2.46	23.8%	2.7%	45.6	78.6%
scan(8e8)	225.04	277.82	4.90	23.5%	3.4%	45.9	79.1%
poly(2e4)	28.21	31.16	0.56	10.5%	4.8%	50.4	86.9%
poly(4e4)	110.94	122.09	2.07	10.1%	-1.8%	53.6	92.4%
poly(8e4)	432.11	469.21	8.05	8.9%	-0.5%	53.7	92.4%
knap(44)	82.01	82.20	1.50	0.2%	5.9%	54.7	94.3%
knap(46)	179.98	181.03	3.21	0.6%	2.9%	56.1	96.7%
knap(48)	575.24	577.32	10.01	0.4%	0.6%	57.5	99.1%
mmul(1024)	69.60	67.41	1.30	-3.1%	11.5%	53.5	92.2%
mmul(1384)	161.33	166.42	2.96	3.1%	3.3%	54.5	94.0%
mmul(2048)	776.09	771.76	13.52	-0.6%	1.6%	57.4	99.0%
abisort(2^{23})	20.01	23.32	3.11	16.5%	684.9%	6.4	11.0%
abisort(2^{24})	42.94	49.81	4.35	16.0%	371.6%	9.9	17.1%
abisort(2^{25})	92.82	106.00	5.98	14.2%	159.5%	15.5	26.7%

Les mesures négatives sont exceptionnelles et signalent un résultat inattendu. Par exemple, le programme parallèle *mmul(1024)* exécuté sur un seul processeur est plus efficace que le programme séquentiel.

TAB. 4.1 – Mesures de performances sur SPARC

nous intéressons surtout au surcoût de gestion, à l'accélération et au surcoût d'initialisation de nos programmes. Comme notre compilateur peut générer deux versions des programmes qu'il compile (optimisé ou non), nous allons comparer leurs performances respectives. Rappelons que la version optimisée d'un programme CPar utilise la convention d'appel de fonction à son avantage afin de réduire le surcoût lié à l'initialisation d'un descripteur de tâche (voir la section 3.8). Les instances des problèmes mesurés sont de plus petite taille que sur SPARC à cause du nombre plus restreint de processeurs physiques. Comme les temps d'exécution sont courts, la variance dans les temps d'exécution peut avoir un impact relatif important sur nos mesures. Afin d'atténuer l'effet de cette variance, nous effectuons une moyenne des mesures sur trente exécutions. Les résultats de nos mesures se trouvent dans les tableaux 4.2 et 4.3. Ces tableaux illustrent, respectivement, les performances des programmes non optimisés et optimisés.

D'abord remarquons qu'il existe une nette différence entre les performances offertes par les programmes optimisés et les autres. Même si les gains de performances tendent à diminuer avec l'accroissement de la granularité, ils restent néanmoins suffisamment significatifs pour justifier l'emploi de l'optimisation. On retrouve cependant des programmes où les différences sont à peine perceptibles. Ensuite, soulignons que les programmes optimisés qui offrent les meilleures performances ne sont pas forcément ceux qui ont la plus importante granularité. Par exemple, les programmes *fib* et *somme* optimisés possèdent tous deux un "overhead" et un surcoût de communication faibles alors que leurs granularités sont fines.

Dans certains cas, un programme parallèle peut être plus efficace qu'un programme séquentiel et ce, même lorsqu'il est exécuté sur un seul processeur. C'est le cas notamment pour le programme *poly* et c'est aussi pourquoi nous obtenons un "overhead" négatif. Nous croyons cependant que ce gain d'efficacité est attribuable à deux facteurs : un meilleur séquencement des opérations lors de l'initialisation du descripteur de tâches et une utilisation plus efficace de la mémoire. Ce genre de résultat n'est donc pas représentatif des performances générales obtenues par le compilateur CPar. En revanche, les programmes moins performants sont *scan* et *abisort*. Ceux-ci at-

Programme	T_s	$T(1)$	$T(4)$	"Overhead" $B(1)$	Surcoût comm. $B_c(4)$	Accél. $S(4)$	Efficacité $E(4)$
fib(28)	0.11	0.13	0.036	18.2%	12.7%	3.05	74.0%
fib(30)	0.28	0.35	0.089	25.0%	2.1%	3.15	76.8%
fib(34)	1.88	2.40	0.602	27.7%	0.4%	3.12	78.0%
somme(4e6)	1.40	1.37	0.355	-2.1%	3.6%	3.94	98.5%
somme(1e7)	2.90	3.47	0.893	19.7%	3.5%	3.25	81.3%
somme(2e7)	5.80	6.94	1.790	19.7%	3.8%	3.24	81.0%
queens(12)	0.50	0.55	0.139	10.0%	1.2%	3.59	89.8%
queens(13)	2.76	3.00	0.754	8.7%	0.6%	3.66	91.5%
queens(14)	16.21	17.72	4.432	9.3%	0.1%	3.66	91.5%
scan(2e6)	1.36	1.60	0.425	17.6%	7.4%	3.20	80.0%
scan(4e6)	2.70	3.31	0.845	22.6%	2.6%	3.20	80.0%
scan(6e6)	3.80	5.09	1.274	33.9%	0.2%	2.99	74.8%
poly(4e3)	0.73	0.70	0.176	-4.1%	0.5%	4.15	103.8%
poly(8e3)	2.92	2.79	0.698	-4.5%	0.1%	4.18	104.5%
poly(2e4)	18.36	17.48	4.351	-4.8%	0.0%	4.21	105.3%
knap(30)	0.29	0.31	0.080	6.9%	3.4%	3.66	91.5%
knap(34)	1.63	1.75	0.441	7.4%	0.1%	3.72	93.0%
knap(36)	6.44	6.91	1.729	7.3%	0.1%	3.72	93.0%
mmul(256)	0.47	0.49	0.125	4.3%	2.1%	3.74	93.5%
mmul(384)	1.55	1.60	0.404	3.2%	1.0%	3.84	96.0%
mmul(512)	3.66	3.75	0.943	2.5%	0.6%	3.88	97.0%
abisort(2^{17})	0.76	0.92	0.254	21.1%	12.6%	3.00	75.0%
abisort(2^{18})	1.66	1.99	0.515	19.9%	4.2%	3.22	80.5%
abisort(2^{19})	3.59	4.33	1.023	20.6%	-0.6%	3.26	81.5%

Les mesures négatives sont exceptionnelles et signalent un résultat inattendu. Par exemple, le programme parallèle *poly(2e4)* exécuté sur un seul processeur est plus efficace que le programme séquentiel.

TAB. 4.2 – Mesures de performances sur Intel sans optimisation

teignent un "overhead" respectif de 18.1% et 19.7% pour leur version optimisée. Nous voyons dans ces deux programmes l'impact négatif de réserver l'utilisation d'un registre pour le pointeur "tail" de notre pile de tâches alors que le nombre de registres sur Intel est déjà restreint. C'est d'ailleurs pourquoi on ne remarquait pas de différences significatives d'"overhead" sur SPARC pour ces deux programmes (voir le tableau 4.1).

Enfin, selon les mesures effectuées, nos programmes CPar possèdent un surcoût de communication généralement faible par rapport à leur taille. Ceci est un atout important étant donné que le surcoût de communication est un facteur limitatif important de l'extensibilité. Ainsi, l'extensibilité des programmes CPar ne se trouve limitée que par un facteur : la portion séquentielle des programmes. Cependant, tous les programmes CPar comportent une partie séquentielle qui consiste à lancer l'exécution des processeurs logiques. Par conséquent le surcoût de communication

Programme	T_s	$T(1)$	$T(58)$	"Overhead" $B(1)$	Surcoût comm. $B_c(4)$	Accél. $S(4)$	Efficacité $E(4)$
fib(28)	0.11	0.11	0.029	0.0%	5.5%	3.70	92.5%
fib(30)	0.28	0.27	0.071	3.7%	5.0%	3.87	96.8%
fib(34)	1.88	1.88	0.474	0.0%	0.9%	3.97	99.3%
somme(4e6)	1.40	1.16	0.302	-17.1%	3.4%	4.64	116.0%
somme(1e7)	2.90	3.02	0.776	4.1%	2.9%	3.74	93.5%
somme(2e7)	5.80	5.98	1.562	3.1%	4.5%	3.72	93.0%
queens(12)	0.50	0.52	0.132	2.9%	2.2%	3.78	94.5%
queens(13)	2.76	2.84	0.715	2.9%	0.7%	3.86	96.5%
queens(14)	16.21	16.78	4.198	3.5%	0.0%	3.86	96.5%
scan(2e6)	1.36	1.41	0.382	3.7%	8.7%	3.56	89.0%
scan(4e6)	2.70	3.04	0.756	12.6%	0.0%	3.57	89.3%
scan(6e6)	3.80	4.49	1.141	18.1%	1.8%	3.33	83.3%
poly(4e3)	0.73	0.70	0.176	-4.1%	0.8%	4.15	103.8%
poly(8e3)	2.92	2.78	0.697	-4.8%	0.3%	4.19	104.8%
poly(2e4)	18.36	17.41	4.345	-5.1%	-0.2%	4.22	105.5%
knap(30)	0.29	0.30	0.076	3.4%	1.9%	3.85	96.3%
knap(34)	1.63	1.66	0.416	1.8%	0.3%	3.92	98.0%
knap(36)	6.44	6.57	1.638	2.0%	-0.2%	3.93	98.4%
mmul(256)	0.47	0.49	0.124	4.3%	1.7%	3.77	94.3%
mmul(384)	1.55	1.60	0.404	3.2%	1.0%	3.84	96.0%
mmul(512)	3.66	3.74	0.943	2.2%	0.9%	3.88	97.0%
abisort(2^{17})	0.76	0.91	0.237	19.7%	5.8%	3.22	80.5%
abisort(2^{18})	1.66	1.95	0.506	17.4%	4.5%	3.28	82.0%
abisort(2^{19})	3.59	4.27	1.079	18.9%	1.4%	3.32	83.0%

Les mesures négatives sont exceptionnelles et signalent un résultat inattendu. Par exemple, le programme parallèle *somme(4e6)* exécuté sur un seul processeur est plus efficace que le programme séquentiel.

TAB. 4.3 – Mesures de performances sur Intel avec optimisation

tend à croître lorsque les instances des problèmes traités sont de plus petite taille parce que le surcoût d'initialisation, qui en fait partie, gagne en importance. Il est clair qu'un programme parallèle dont le court temps d'exécution ne permet pas au noyau de lancer (et de faire travailler) tous les processeurs logiques n'affichera pas de bonnes performances.

Ces tests ont mis en évidence les gains de performance réalisables lorsque l'on prend avantage des caractéristiques de la plateforme. Ainsi, on remarque que la stratégie de compilation optimisée pour la plateforme Intel possède l'avantage de réduire considérablement le surcoût de gestion relié à la création des tâches qui est souvent la principale source de dégradation des performances.

4.7 Tests sur l'algorithme génétique

Le programme *tsp* ne peut pas être évalué de la même manière que les autres programmes, car il n'y a pas que le temps d'exécution qui compte, mais aussi la qualité des solutions générées, et le nombre de générations nécessaires à l'obtention de ces solutions. Ainsi, nous proposons des tests qui mettront en évidence ces deux caractéristiques des exécutions du programme *tsp*. Les résultats pour la plateforme Intel sont illustrés dans le tableau 4.4.

Dans ce tableau, "Pop." fait référence à au nombre de solutions candidates qui sont conservées d'une génération à l'autre. La taille de la population possède un impact important sur les performances du programme. Si la population est trop faible, l'algorithme converge rapidement vers une solution généralement moins bonne. À l'opposé, une population plus importante permet de converger vers une meilleure solution, mais au détriment d'un temps de calcul plus long à chaque itération. "Gen." correspond au nombre de générations produites par le programme et fournit ainsi une autre mesure de l'effort total requis par l'exécution du programme. En général, un nombre de générations inférieur est préférable. "%opt." est une mesure de la qualité des solutions produites. Elle est le rapport entre la longueur du chemin de la meilleure solution et la longueur du chemin optimal connu (la solution optimale est

Exécutions des programmes séquentiels

Programme	Pop.	T_s	Gen.	% opt.*	Sol. opt.
tsp(pr246)	113	21.5	15.7	0.059%	7 sur 10
tsp(pcb442)	110	120.4	31.5	0.37%	0 sur 10
tsp(u1060)	212	1894.0	68.2	0.57%	0 sur 10

Exécutions des programmes parallèles

Programme	Pop.	$T(4)$	Gen.	% opt.*	Sol. opt.	Accélération
tsp(pr246)	113	6.8	18.1	0.063%	7 sur 10	3.16
tsp(pcb442)	110	32.0	33.9	0.53%	0 sur 10	3.76
tsp(u1060)	212	415.0	59.0	0.59%	0 sur 10	4.56

Les mesures exprimées sont des moyennes pour dix exécutions.
* $\%\text{opt} = \left(\frac{Solution\,obtenue}{Solution\,optimale} - 1\right) \times 100\%$.
Une valeur de 0% signifie que la solution optimale est atteinte.

TAB. 4.4 – Test du programme *tsp* sur Intel

en effet connue pour toutes les instances testées). Par exemple, une solution dont la longueur est de 101 alors que la solution optimale est de 100 est à 1% de l'optimale ($\frac{101}{100} - 1$). Enfin, "Sol. opt" note le nombre de fois qu'il a été possible d'obtenir la solution optimale.

Comme on peut le constater, les solutions apportées par le programme parallèle sont légèrement moins bonnes que celles proposées par le programme séquentiel ce pour un nombre de générations comparable. Cette différence dans la qualité des solutions produites s'explique par le caractère aléatoire de l'algorithme. On obtient toutefois des accélérations acceptables qui se situent entre 3.16 et 4.56. Notons que même si toutes les solutions candidates d'une génération sont créées en parallèle, leur sélection ultérieure, une fois toutes construite (qui s'effectue au moyen d'un tri), et leur migration vers la génération suivante, se font de manière séquentielle et a pour conséquence de réduire les performances du programme parallèle.

4.8 Tests de comparaison avec ParSubC

L'objectif de CPar est d'obtenir des performances supérieures à celles de ParSubC. Il convient donc d'effectuer certains tests comparatifs entre les programmes générés par ces deux compilateurs. A cette fin, nous avons repris certains programmes et les avons compilés avec ParSubC sur la plateforme SPARC. Le résultat de cette

Programme		ParSubC		CPar	
	T_s	$T(1)$	"Overhead" $B(1)$	$T(1)$	"Overhead" $B(1)$
fib(36)	2.08	15.1	626%	2.83	36%
queens(13)	5.38	16.0	197%	6.46	20%
sum(4e7)	4.73	14.0	196%	5.91	25%
poly(8e3)	5.00	8.25	65%	4.67	-7%

TAB. 4.5 – Comparaison des performances de ParSubC et de CPar

comparaison se trouve au tableau 4.5. On constate que les performances de CPar sont nettement meilleures (d'un ordre de grandeur) que celles de ParSubC. Ces résultats viennent donc appuyer nos choix d'implantation ainsi que notre stratégie de compilation. Notons cependant que CPar ne peut pas distribuer le calcul sur un ensemble d'ordinateurs comme le fait ParSubC. Ainsi, CPar évite de consacrer des ressources de calcul considérables à l'aspect distribué de la gestion des tâches.

4.9 Tests de comparaison avec Cilk

Nous avons effectué des tests servant à comparer les performances de CPar avec celle de Cilk 5.2 sur la plateforme Intel. Ces tests ne sont effectués que sur la plateforme Intel car les outils nécessaires à la compilation de Cilk n'étaient pas disponibles sur le Sun Enterprise 10000.

Comme on peut le remarquer par les résultats des tests qui se trouvent au tableau 4.6, CPar donne des performances qui sont systématiquement supérieures à celles de Cilk. Notons cependant que les différences diminuent lorsque la granularité des tâches croît. Par exemple, pour une fonction à granularité fine comme Fibonacci, CPar atteint une accélération quasi linéaire alors que Cilk n'arrive pas à atteindre les performances du programme séquentiel avec quatre processeurs. En revanche, pour un programme à granularité plus grande comme la multiplication de polynômes, les résultats sont plus proches : des accélérations de 4.19 contre 3.70 pour CPar et Cilk respectivement.

Il est intéressant de constater que la différence au niveau des performances brutes n'est pas causée principalement par le surcoût de communication. Elle semble plu-

Prog. CPar	T_s	$T(1)$	$T(4)$	"Overhead" $B(1)$	Surcoût comm. $B_c(4)$	Accélération $S(4)$	Efficacité $E(4)$
fib(34)	1.88	1.88	0.474	0.0%	0.9%	3.97	99.3%
somme(4e6)	1.40	1.16	0.302	-17%	3.4%	4.64	116%
queens(13)	2.76	2.84	0.715	2.9%	0.7%	3.86	96.5%
scan(4e6)	2.70	3.04	0.756	12.6%	0.0%	3.57	89.3%
poly(8e3)	2.92	2.78	0.697	-4.8%	0.3%	4.19	105%
knap(34)	1.63	1.66	0.416	1.8%	0.3%	3.92	98.0%
mmul(384)	1.55	1.60	0.404	3.2%	1.0%	3.84	96.0%
Prog. Cilk	T_s	$T(1)$	$T(4)$	"Overhead" $B(1)$	Surcoût comm. $B_c(4)$	Accélération $S(4)$	Efficacité $E(4)$
fib(34)	1.88	8.27	2.12	340%	11.2%	0.89	22.3%
somme(4e6)	1.40	4.24	1.11	203%	14.3%	1.26	31.5%
queens(13)	2.76	5.94	1.52	115%	5.1%	1.80	45.0%
scan(4e6)	2.70	9.47	2.42	240%	26.5%	1.12	28.0%
poly(8e3)	2.92	2.90	0.772	-0.7%	6.4%	3.78	94.5%
knap(34)	1.63	2.61	0.696	60%	10.6%	2.34	58.5%
mmul(384)	1.55	2.09	0.562	35%	10.2%	2.76	69.0%

Les mesures négatives sont exceptionnelles et signalent un résultat inattendu. Par exemple, le programme parallèle *somme(4e6)* exécuté sur un seul processeur est plus efficace que le programme séquentiel.

TAB. 4.6 – Comparaison entre CPar et Cilk sur Intel

tôt être attribuable à l'"overhead". En effet, les résultats des tests montrent qu'il existe presqu'un ordre de grandeur de différence entre les surcoûts de communication de CPar et ceux de Cilk. Cependant, ces mêmes tests nous montrent aussi qu'il y a jusqu'à deux ordres de grandeur entre les "overheads" des deux systèmes. Cette constatation n'est pas trop surprenante pour deux raisons : premièrement, parce que CPar emploie les mêmes idées de fond que Cilk, et deuxièment, parce que le compilateur CPar n'applique que des transformations locales alors que Cilk transforme complètement les fonctions contenant des appels parallèles.

La stratégie de compilation de CPar semble non seulement plus efficace mais lui permet aussi d'exploiter certaines caractéristiques de l'architecture sous-jacente afin de minimiser le surcoût de gestion. Il est clair que du point de vue de la vitesse d'exécution des programmes à granularité fine, CPar possède un avantage net sur Cilk.

Chapitre 5

Conclusion

CPar est un langage de programmation simple et puissant. Il permet d'exprimer facilement le parallélisme de contrôle dans un programme. Les autres langages du genre sont souvent complexes et leurs implantations trop peu performantes pour le calcul parallèle d'une granularité fine. Pour combler ce vide, l'objectif principal de notre compilateur CPar consiste à minimiser le surcoût de gestion engendré par l'utilisation du parallélisme sur la vitesse d'exécution des programmes CPar. Afin d'atteindre cet objectif, CPar reprend les principes de ParSubC et en refait une nouvelle implantation.

Le compilateur CPar génère du code C dans le dialecte de *gcc* et ne modifie le code qu'aux endroits où les constructions d'appels parallèles existent. Les étapes importantes dans l'optimisation de la génération de code sont les suivantes :

- Employer une représentation efficace de la pile de tâches.
- Trouver une technique efficace pour la création et l'initialisation des descripteurs de tâches.
- Déterminer la stratégie d'accès aux données locales d'un processeur.
- Implanter efficacement l'empilement et le dépilement d'une tâche.
- Effectuer un balancement automatique de la charge de travail en utilisant la technique du vol de tâches.
- Mettre au point une interface simple pour l'exécution d'une tâche via les fonctions

"proxy".

- Implanter un mécanisme simple mais efficace de synchronisation entre les tâches.
- Modifier le protocole d'exclusion mutuelle rapide afin d'éviter l'utilisation d'écriture en mémoire de type "write-through".

Ce sont les tests empiriques qui déterminent l'efficacité de l'implantation, et par conséquent si notre objectif est atteint ou pas. La série de tests effectués démontrent que nous obtenons un surcoût de gestion raisonnablement faible sur la plateforme SPARC, mais que sur Intel les performances sont encore meilleures. Le surcoût de gestion peut être réduit considérablement lorsque l'on prend avantage de l'architecture sous-jacente. CPar permet donc au programmeur d'exploiter le parallélisme dans ses programmes à un coût relativement faible. Le problème de la granularité s'en trouve aussi diminué puisque CPar supporte très bien la création d'un grand nombre de tâches d'une granularité fine. Il peut ainsi lui offrir un balancement automatique et équitable de la charge de travail.

Il serait cependant possible d'améliorer les performances de CPar en l'incorporant à même le compilateur *gcc*. Cette approche pourrait d'une part réduire les coûts d'initialisation des descripteurs pour la plateforme SPARC en s'inspirant de l'optimisation Intel. D'autre part, il serait possible pour la plateforme Intel de laisser les arguments s'accumuler sur la pile comme *gcc* le fait normalement. Le compilateur pourrait aussi faire une gestion plus intelligente du registre dédié au pointeur "tail". De plus, CPar pourrait bien s'inspirer de Cilk et de StackThreads afin d'offrir au programmeur un ensemble de primitives de synchronisations tel que des "mutex", des sémaphores ainsi que des compteurs atomiques.

Bibliographie

[1] Peter B. Galvin Abraham Silberschatz. *Operating Systems Concepts.* Addisson Wesley, fourth edition, 1995.

[2] Anonyme. An introduction to PVM programming. World-Wide Web, 1996.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall et Yuli Zhou. Cilk : An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1) :55–69, 25 August 1996.

[4] David R. Butenhof. *Programming with POSIX threads.* Addison-Wesley, Reading, MA, USA, 1997.

[5] Ismaïl Chabini et Bernard Gendron. Parallel performance measures revisited. In Vincent Van Dongen, editor, *Proceedings of the High Performance Computing Symposium '95, Canada's Ninth Annual International High Performance Computing Conference and Exhibition*, pages 381–392, Montréal, Québec, July 10–12, 1995. Centre de Recherche Informatique de Montréal.

[6] Jack J. Dongarra, Steve W. Otto, Marc Snir et David Walker. An introduction to the MPI standar. Technical Report CS-95-274, University of Tennessee, January 1995.

[7] Hesham El-Rewini et Ted G. Lewis. *Distributed and Parallel Computing.* Manning, 3 Lewis Street, Greenwich CT 06830, 1998.

[8] M. Feeley. Lazy remote procedure call and its implementation in a parallel variant of C. *Lecture Notes in Computer Science*, 1068, 1996.

[9] Marc Feeley. An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors. Technical Report IRO-869, Dept. d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1993.

[10] Chris W. Fraser et David R. Hanson. *A Retargetable C Compiler : Design and Implementation.* Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[11] Matteo Frigo, Charles E. Leiserson et Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Canada, 17–19 June 1998.

[12] Borland International Inc. *Turbo Assembler 3.0 Quick Reference Guide.* Borland International, 18000 Green Hill Road, P.O. Box 660001, Scotts Valeey CA 95067-0001, 1991.

[13] Francis L'Ecuyer. Conception et réalisation d'une variante parallèle de C basée sur la création paresseuse de tâches. Master's thesis, Dept. d'Informatique et de Recherche Opérationnelle, Université de Montréal, dec 1997.

[14] Colin R. Reeves. *Modern Heuristic Techniques for Combinational Problems.* McGraw-Hill Book Company, Berkshire, 1995.

[15] Gerhard Reinelt. Tsplib. World-Wide Web : http ://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/index.html, 1995.

[16] Supercomputing Technology Group MIT Laboratory for Computer Science. *Cilk 5.2 Reference Manual*, jul 1998. MIT.

[17] Andrew S. Tanenbaum. *Modern Operating Systems.* Prrentice-Hall, 1992.

[18] Kenjiro Taura, Kunio Tabata et Akinori Yonezawa. StackThreads/MP : Integrating futures into calling standards. In A. Andrew Chien et Marc Snir, editors, *Proceedings of the 1999 ACM Sigplan Symposium on Principles and Practise of Parallel Programming (PPoPP'99)*, volume 34.8 of *ACM Sigplan Notices*, pages 60–71, A.Y., May 4–6 1999. ACM Press.