Université de Montréal

Implantation des Futures sur un Système Distribué par Passage de Messages

par
Jérémie Lasalle Ratelle

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

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

Août, 2010

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé:

Implantation des Futures sur un Système Distribué par Passage de Messages

présenté par:

Jérémie Lasalle Ratelle

a été évalué par un jury composé des personnes suivantes:

El Mostapha Aboulhamid,	président-rapporteur
Marc Feeley,	directeur de recherche
Bruno Dufour,	membre du jury

Mémoire accepté le: .....

RÉSUMÉ

Ce mémoire présente une implantation de la création paresseuse de tâches destinée à des systèmes multiprocesseurs à mémoire distribuée. Elle offre un sous-ensemble des fonctionnalités du Message-Passing Interface et permet de paralléliser certains problèmes qui se partitionnent difficilement de manière statique grâce à un système de partitionnement dynamique et de balancement de charge. Pour ce faire, il se base sur le langage Multilisp, un dialecte de Scheme orienté vers le traitement parallèle, et implante sur ce dernier une interface semblable à MPI permettant le calcul distribué multiprocessus. Ce système offre un langage beaucoup plus riche et expressif que le C et réduit considérablement le travail nécessaire au programmeur pour pouvoir développer des programmes équivalents à ceux en MPI. Enfin, le partitionnement dynamique permet de concevoir des programmes qui seraient très complexes à réaliser sur MPI. Des tests ont été effectués sur un système local à 16 processeurs et une grappe à 16 processeurs et il offre de bonnes accélérations en comparaison à des programmes séquentiels équivalents ainsi que des performances acceptables par rapport à MPI. Ce mémoire démontre que l'usage des futures comme technique de partitionnement dynamique est faisable sur des multiprocesseurs à mémoire distribuée.

Mots clés : Langages de programmation fonctionnels, Scheme, Multilisp, Futures, Traitement parallèle, Traitement distribué, création paresseuse de tâches

ABSTRACT

This master's thesis presents an implementation of lazy task creation for distributed memory multiprocessors. It offers a subset of Message-Passing Interface's functionality and allows parallelization of some problems that are hard to statically partition thanks to its dynamic partitionning and load balancing system. It is based on Multilisp, a Scheme dialect for parallel computing, and implements an MPI like interface on top of it. It offers a richer and more expressive language than C and simplify the work needed to developp programs similar to those in MPI. Finally, dynamic partitioning allows some programs that would be very hard to develop in MPI. Tests were made on a 16 cpus computer and on a 16 cpus cluster. The system gets good accelerations when compared to equivalent sequential programs and acceptable performances when compared to MPI. It shows that it is possible to use futures as a dynamic partitioning method on distributed memory multiprocessors.

Keywords: Functional programming languages, Scheme, Multilisp, Futures, parallel computing, distributed computing, lazy task creation

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

REMERCIEMENTS

Je dédie ce mémoire à Rolland Ratelle et Léonard Lasalle.

Je tiens tout d'abord à remercier mon Directeur de recherche, Marc Feeley, sans qui ce mémoire n'aurait pas été possible. Il a développé les idées qui sont à l'origine des travaux présentés et il a eu une profonde influence sur ma vision de l'informatique.

Je tiens aussi à remercier Guillaume Cartier qui m'a permi de faire mes preuves comme programmeur en m'offrant mon premier emploi en informatique à l'été 2007.

Je remercie sincèrement mes parents, Monique et Dominique, qui m'ont supporté tout au long de mes études. Ils ont toujours cru en moi et ils vont enfin pouvoir savourer notre réussite. J'apprécie toujours le temps passé avec vous.

Je remercie mon frère Fabrice avec qui j'ai cohabité pendant mes études. Il a su tolérer mes côtés moins agréables et je lui souhaite beaucoup de succès.

Je remercie ma copine Audrey ainsi que ses parents, Nicole et Richard, qui m'ont beaucoup encouragé pendant les derniers mois de rédaction alors que le stress s'était emparé de moi.

Je veux remercier mon ami Benjamin Provencher, mon complice tout au long de mes études. Nous avons partagé de très bons moments et de multiples discussions dont plusieurs en informatiques m'ont apporté des perspectives différentes. J'espère qu'on se verra encore souvent pour ricaner autour d'une bonne stout.

Je tiens a remercier mon cousin Jean-François Ratelle, mon partenaire de délires et de fabulations. Nos soirées bien arrosées et complètement démentes ainsi que celles moins explosives à divaguer devant un bon feu de bois m'ont offert des moments où je pouvais complètement décrocher.

Enfin je veux remercier mon collègue et ami David Haguenauer, avec qui j'ai partagé le laboratoire de traitement parallèle pendant près de six ans. Merci d'avoir ecouté et su rire de toutes mes histoires folles. Ca va bien me manquer.

CHAPITRE 1

INTRODUCTION

Le parallélisme est aujourd'hui plus pertinent que jamais. En effet, avec le développement récent des processeurs multicoeurs et le rapprochement des limites à la loi de Moore, il s'avère qu'il n'est plus seulement limité au calcul scientifique haute performance comme cela avait été le cas pendant plusieurs années. De plus, la création d'énormes applications réseau sur internet a aussi stimulé l'intérêt dans les techniques de programmation distribuée à grande échelle qui étaient autrefois principalement utilisées dans les applications contrôlant les réseaux de télécommunication.

Alors que le calcul scientifique parallèle a été dominé pendant plusieurs années par les gros multiprocesseurs à mémoire partagée, on voit dans la dernière décennie une tendance à l'adoption de systèmes à mémoire distribuée de type *grappe* composés d'un grand nombre d'unités à faible coût connectées par des réseaux rapides. Ces systèmes offrent l'avantage de ne pas être limités par l'architecture d'un système unique, mais de pouvoir passer beaucoup mieux à l'échelle. Toutefois, ces systèmes ne présentent pas que des avantages. En effet, ils sont extrêmement complexes à concevoir et à programmer et cette complexité ne cesse de grandir.

1.1 Motivation

Le modèle dominant actuel de programmation pour ces systèmes ne fait rien pour améliorer le problème. En effet, il combine l'utilisation d'un langage de programmation de bas niveau, comme C, avec la librairie *Message Passing Interface* [21]. En plus de laisser au programmeur les responsabilités inhérentes à un langage offrant peu d'abstractions, ce modèle le contraint à distribuer manuellement les structures de données globales aux algorithmes vers l'entreposage local aux noeuds, à contrôler explicitement les communications ainsi que leurs protocoles, à partitionner statiquement les tâches à accomplir ainsi qu'à contrôler le balancement de charge.

Pour certains problèmes simples, ces tâches peuvent s'avérer relativement faciles. Mais dans la plupart des cas, les applications développées sur les grandes machines à mémoire distribuée sont d'une énorme complexité et le modèle existant, en obligeant le programmeur à se soucier de problèmes qui ne sont pas directement en lien avec la tâche à accomplir, augmente la quantité d'erreurs, le temps de développement et la difficulté à maintenir les programmes.

De plus, lorsque la complexité augmente et que plusieurs modules interagissent, surtout s'ils sont de natures symboliques, l'absence d'un système de sérialisation, obligeant un encodage manuel des communications, devient une limitation non négligeable. Un exemple d'un tel système serait un compilateur parallèle travaillant sur des structures chainées, des arbres et des tables de symboles, représentant le code. Cela rend très difficile la réutilisation de modules déjà existants faisant usage de parallélisme, car le programme doit connaitre de manière dynamique la représentation des données.

Un autre problème est l'absence de partitionnement et de distribution dynamique du travail. Le programmeur doit être en mesure de décider au moment de l'écriture de l'application quelles tâches rouleront sur quels systèmes à tout moment. Il est évident que l'information quant au nombre de processus étant disponible à l'exécution, il est possible d'effectuer un partitionnement dynamique naïf lorsqu'un problème se décompose facilement en sous-problèmes de tailles sensiblement égales. Mais aussitôt que la grosseur des sous-tâches devient variable ou bien que le système possède plusieurs modules opérant de manière concurrente, il sera très difficile de rendre optimale l'utilisation des ressources.

Une des façons de contrer ces problèmes est d'adopter une approche où certains processus servent de distributeurs de travail. Ils maintiennent une liste de travail à accomplir et fournissent les autres processus lorsque ceux-ci en ont besoin. Cela permet d'avoir différentes sous-tâches de tailles variables et ne pas perdre de ressources sur les systèmes les effectuant. Toutefois, on doit à ce moment assumer la perte de ressources pour les processus se chargeant uniquement de la distribution.

MPI n'est pas non plus à l'abri du problème de la granularité des tâches. Comme la création et la distribution de ces dernières possèdent un coût, on doit en tenir compte lorsqu'on prend la décision de paralléliser ou non du travail. Si cette décision doit être prise par le programmeur lors de la conception, il devra alors pencher vers un niveau de granularité donné. Dépendant de son choix et du système sur lequel l'application roulera il se peut qu'il y ait des pertes de parallélisme potentiel. Si le grain s'avère trop fin, le surcoût sera trop élevé et on perdra en performances absolues.

Enfin, un autre des problèmes de *MPI* est la présentation au programmeur d'un modèle partitionné du programme. C'est une approche au parallélisme dans laquelle le programmeur doit construire ses algorithmes en séparant explicitement ceux-ci entre plusieurs tâches. Ce dernier s'oppose au modèle global où le programmeur introduit le parallélisme à l'aide de primitives abstraites du langage qui effectuent automatiquement la distribution du travail et des données.

Pour conclure, le modèle de programmation actuel des machines à mémoire distribuée, qui combine le langage *C* avec la librairie *MPI* souffre de plusieurs problèmes qui rendent son utilisation difficile :

1. Utilisation de *C*, un langage de bas niveau qui laisse beaucoup de responsabilités au programmeur, comme la gestion de la mémoire.

2. Absence d'un système de sérialisation forçant ainsi un encodage statique des données.

3. Absence de partitionnement et de redistribution dynamique des tâches.

4. Présence du problème de granularité des tâches.

5. Complexité du modèle partitionné.

1.2 Solution

Comme la complexité des algorithmes et des plates-formes matérielles distribuées va continuer d'augmenter, l'utilisation des systèmes de bas niveau deviendra de moins

en moins réaliste pour les programmeurs. Il devient vite nécessaire d'adapter les techniques d'abstraction conçues pour le traitement parallèle en mémoire partagée ainsi que d'en développer de nouvelles qui permettront d'augmenter la productivité des programmeurs.

Certaines de ces techniques, comme la *Création Paresseuse de Tâches* [13] souvent considérée comme la meilleure approche à l'implantation du partage de charge dynamique, ont déjà fait leurs preuves sur des systèmes à grande échelle. D'autres, comme la mémoire partagée distribuée et la mémoire transactionnelle ont fait l'objet de beaucoup de recherche dans les dernières années. Finalement, la plupart des nouveaux langages développés en informatique haute performance explorent l'idée de l'espace d'adressage global partitionné et permettent de considérer la localité des données lors du balancement de charge.

Dans ce mémoire, nous tentons de démontrer que l'usage d'un langage de programmation de haut niveau, *Multilisp*, équipé du partage dynamique de charge par vol de tâches, permet de faciliter énormément le travail du programmeur en lui évitant tous les problèmes associés à l'usage d'un langage de bas niveau comme *C*, comme la gestion manuelle de la mémoire, ainsi que les difficultés du partitionnement statique des tâches et du contrôle manuel du partage de charge. Nous montrons qu'il est possible d'implanter et d'utiliser le partage dynamique de charge dans un langage offrant un modèle partitionné et qu'il permet tout de même une grande simplification des programmes. Notre système se veut un éventuel tremplin vers un langage à modèle global, mais n'offre pas, pour l'instant, l'ensemble des outils requis par un tel système. Nous démontrons aussi qu'il est possible d'obtenir de très bonnes performances dans un langage offrant de telles abstractions. Pour ce faire, nous développons un système hybride, qui unifie le langage *Multilisp* avec des primitives de communication inspirées d'*Erlang* et *MPI*. Il implante le partage dynamique de charge à l'aide de la *Création Paresseuse de Tâches*.

1.3 Survol

Dans le chapitre deux, nous introduisons les différents concepts pertinents relatifs au calcul parallèle et distribué. Nous abordons notamment les différents paradigmes de programmation parallèle ainsi que les variantes d'architectures matérielles les permettant. Nous présentons ensuite les deux systèmes de programmation parallèle dont ce mémoire s'inspire, c'est-à-dire *Multilisp* et *MPI*.

Dans le chapitre trois, nous abordons le sujet des *continuations*. Nous introduisons les concepts généraux pertinents et fournissons des exemples d'utilisation. Par la suite, nous expliquons en détail la manière dont elles sont implantées efficacement dans le compilateur *Gambit-C* [12].

Dans le chapitre quatre, nous expliquons en détail l'implantation du système à l'aide du langage de programmation *Termite* [14] conçu sur le compilateur *Gambit-C*. Nous mettons surtout l'emphase sur l'implantation de la *Création Paresseuse de Tâche* à l'aide des continuations ainsi que du protocole de communication permettant la distribution dynamique des tâches.

Enfin dans le chapitre cinq, nous présentons les résultats de notre expérimentation. Nous montrons avec plusieurs exemples de programmes la simplicité de développement d'applications dans notre système et analysons aussi la performance de ces derniers face à des équivalents programmés en *MPI* et *Erlang*.

CHAPITRE 2

CONNAISSANCES

À l'origine, le traitement informatique utilisait une approche séquentielle. Les tâches à accomplir étaient divisées en une série d'instructions qui devait être exécutée par une unité logique. Le travail de chacune était accompli avant que le processeur en entame une autre.

Toutefois, avec le développement de programmes plus grands et plus gourmands en ressources informatiques, notamment dans les domaines scientifiques et du génie où des calculs énormes doivent être accomplis sur de grands ensembles de données, il était nécessaire de trouver une manière de pouvoir traiter ces tâches plus rapidement que sur les machines séquentielles les plus rapides.

En effet, certains de ces problèmes sont si gros qu'ils représentent un coût énorme en temps et peuvent même parfois être techniquement impossibles à compléter sur un ordinateur séquentiel. C'est pourquoi la programmation parallèle a été développée. C'est une approche au traitement informatique où l'on utilise plusieurs ressources travaillant en même temps pour résoudre une tâche donnée.

Pour ce faire, on doit trouver des façons de diviser le travail en plusieurs sous-tâches qui pourront être accomplies de façon concurrente par les ressources présentes. La manière de faire cette division correctement et d'assigner le travail résultant est le problème fondamental du traitement parallèle.

Il existe beaucoup de manières d'effectuer du calcul parallèle, autant au niveau des techniques que de la configuration du matériel utilisé et de la manière de créer les programmes. Dans ce chapitre, nous présenterons tout d'abord les différents types de parallélisme. Ensuite, nous ferons un survol des organisations matérielles utilisées pour faire le traitement parallèle, dont les systèmes multiprocesseurs à mémoire partagée et les systèmes distribués. Nous étudierons en détail une approche adaptée à chacun de ces systèmes, soit *Multilisp* et le *passage de messages*. Dans la section sur le *passage*

de messages nous parlerons du *Message Passing Interface* ainsi que des *systèmes acteur* comme *Erlang* [26] et *Termite* [14]. Nous indiquerons les problèmes que chacune de ces approches tente de régler dans son environnement respectif et finalement nous démontrerons comment nous utilisons les concepts de Multilisp afin de pouvoir utiliser le *partitionnement dynamique* dans un système à passage de messages sur des machines distribuées.

2.1 Types de parallélisme

Il existe plusieurs types de parallélisme et mécanismes permettant le parallélisme. Les types de parallélisme sont en général définis en fonction de la relation entre le calcul à effectuer et les données sur laquelle il doit être fait. Les mécanismes quant à eux sont plus définis d'après la relation de granularité entre les différentes unités de calcul.

Un concept important en parallélisme est le *passage à l'échelle*. Il représente la capacité d'un système d'accommoder une plus grande quantité de travail tout en maintenant son efficacité, c'est-à-dire l'utilisation optimale de ses unités de calcul pour du travail utile. Il existe donc en fait deux facettes à cette caractéristique. Tout d'abord la capacité du système de minimiser la quantité de travail de gestion malgré l'augmentation de la quantité de travail. Ensuite, la manière dont les algorithmes parallèles utilisés peuvent s'adapter à une quantité de données plus grande. Un passage à l'échelle idéal permettra d'obtenir un facteur d'accélération de N lorsqu'on utilise N fois plus de ressources matérielles (CPUs, mémoires, etc.).

Le premier type de parallélisme est le parallélisme d'instructions qui consiste à exécuter plusieurs instructions ou parties d'instructions d'un programme de manière concurrente à l'intérieur d'un même processeur. Il existe deux grands mécanismes qui le rendent possible. Tout d'abord, le *pipelining d'instructions* est une technique où on divise l'exécution d'une instruction en plusieurs étapes distinctes qui sont prises en charge par des modules matériels distincts à l'intérieur du processeur. Cela rend possible une exécution concurrente de ces étapes et donc une superposition des différentes parties

de plusieurs instructions.

L'autre mécanisme, l'architecture dite *superscalaire* consiste à multiplier les unités logiques à l'intérieur du processeur pour lui permettre d'exécuter plusieurs instructions pareilles de manière simultanée. Par exemple, on équipera le processeur de deux additionneurs ce qui permettra, s'il n'existe pas de relation de dépendance entre les données, d'exécuter deux additions successives en même temps dans le processeur.

Un autre type de parallélisme est le parallélisme de données. Il survient lorsqu'on désire effectuer des opérations sur un ensemble de données pouvant être divisé en plusieurs parties et que le travail sur chacune de ces parties peut être fait de manière concurrente et indépendante. Ce type de parallélisme passe très bien à l'échelle, car l'augmentation de la taille des données offre de plus grandes possibilités de parallélisme. Un exemple de parallélisme de données au niveau de l'architecture est le *Single Instruction Multiple Data*, qui est un ordinateur où plusieurs unités de calcul exécutent le même fil d'instruction sur des données différentes.

Finalement, le dernier type de parallélisme est le parallélisme de tâches. Il consiste à exécuter plusieurs parties différentes d'un programme en même temps lorsque ces parties ne sont pas dépendantes. À ce moment, chaque unité de calcul exécute un code différent des autres. C'est ce type de parallélisme qui possède la plus grande granularité. Ce type de parallélisme passe mal à l'échelle, car la structure de l'algorithme détermine directement les possibilités d'accélération. Cependant, il est nécessaire pour certains calculs dits symboliques où il n'y a pas de parallélisme de données.

2.2 Matériel Parallèle

Dans cette section, nous allons présenter les caractéristiques des deux grandes classes d'architectures parallèles. Il s'agit des architectures à mémoire partagée et des architectures à mémoire distribuée.

2.2.1 Architectures à mémoire partagée

Les architectures à mémoire partagée aussi connues en tant que *SMP* qui signifie *Symmetric Multiprocessor* consistent en des ordinateurs possédant plusieurs processeurs identiques étant connectés à une mémoire centrale partagée. Tous les processeurs ont accès à cette mémoire, mais travaillent de manière indépendante et concurrente. Un changement effectué par un processeur à la mémoire est visible aux autres processeurs.

Il existe cependant deux approches à l'organisation de la mémoire dans ces architectures. Les architectures à accès mémoire uniforme ne possèdent qu'une seule mémoire et tous les processeurs y sont connectés de la même manière. On les appelle uniforme, car le temps d'accès à la mémoire est le même pour tous les processeurs du système. Ces architectures ne sont plus très utilisées aujourd'hui, ayant été grandement remplacées par les architectures à accès mémoire non uniformes.

Dans ces dernières, les processeurs sont organisés en banques indépendantes les unes des autres. Chaque banque possède une mémoire locale dont l'accès est très rapide pour les processeurs faisant partie de cette dernière. On appelle ces architectures non uniformes car les processeurs ont accès à l'ensemble des banques de mémoire, mais le temps d'accès aux différentes banques n'est pas le même. Les architectures modernes où les processeurs possèdent une mémoire cache privée sont un exemple de système à accès mémoire non uniforme.

L'objectif des architectures à accès non uniforme est de régler le problème de bande passante de la mémoire. En effet, les processeurs modernes sont beaucoup plus rapides que la mémoire centrale et dans un système possédant un grand nombre de processeurs accédant tous à la même mémoire, les processeurs peuvent se trouver à passer la plus grande partie de leur temps à attendre après la mémoire au lieu de travailler.

2.2.2 Architectures à mémoire distribuée

Les architectures à mémoire distribuée consistent en un ensemble de systèmes possédant chacun leur mémoire locale privée qui est inaccessible par les autres systèmes.

Ils sont reliés entre eux par un réseau de communication qui peut être parfois un réseau spécialisé très rapide tel que *Gigabit Ethernet* [2] et *Infiniband* [4], ou au contraire être un réseau lent et peu robuste comme Internet.

Comme la mémoire de chaque système est privée, les processeurs ne peuvent accéder à la mémoire des autres. Pour cette raison, une tâche du programmeur, lorsqu'il développe pour de tels systèmes, est de définir comment et dans quelles circonstances les processeurs devront communiquer pour échanger des données ou de l'informations.

L'avantage de ces systèmes est qu'ils passent bien à l'échelle au niveau du nombre de processeurs et de la quantité de mémoire. La bande passante du réseau de communication pourra par contre être un problème de taille. Ces systèmes peuvent aussi être construits à l'aide d'ordinateurs de consommation et ne nécessitent pas de matériel hautement spécialisé et dispendieux ce qui réduit donc les coûts.

Les systèmes composant une architecture à mémoire distribuée peuvent eux-mêmes être des ordinateurs multiprocesseurs à mémoire partagée. Ceux-ci forment en fait la majeure partie des superordinateurs qui sont aujourd'hui utilisés pour effectuer les calculs scientifiques d'envergure.

2.3 État de l'art

Les langages et les outils pour le développement d'applications parallèles ont fait l'objet d'énormément de recherche lors des 20 dernières années et le domaine est toujours en pleine expansion. Une partie importante de ce travail concerne les techniques permettant l'automatisation de la distribution du travail, le balancement de charge dynamique.

Dans cette section, nous ferons la différence entre les langages fournissant un modèle *fragmenté* de parallélisme et ceux offrant un modèle *global* [8].

Le modèle *fragmenté*, plus précisément le modèle *programme unique données multiples*, force le programmeur à concevoir son application autour du concept de tâches associées à un processus particulier et à faire dépendre l'algorithme ainsi que la forme des

données de cette limitation dans l'expression du parallélisme. Par exemple, le programmeur devra explicitement diviser les données du programme en morceaux en fonction du nombre de processus présents et se charger de distribuer et contrôler le travail de chacun de manière explicite. Dans ce modèle, on exécute habituellement une copie du programme sur chaque processeur faisant partie du système et le niveau de granularité du parallélisme est donc limité au nombre de processus. C'est le modèle qui domine sur les machines distribuées, car il est le plus facile à implanter laissant beaucoup de responsabilités au programmeur et réduisant donc la complexité des compilateurs et des librairies. Il est aussi assez naturel à cause de la structure des ordinateurs distribués qui sont constitués d'une grande quantité de noeuds indépendants.

Le modèle *global* quant à lui offre une vision unifiée du programme. Du point de vue du programmeur, une seule copie de son programme est exécutée et le parallélisme y est exprimé à l'aide de primitives et d'abstractions du langage. Cela permet d'exprimer de manière beaucoup plus naturelle et concise les algorithmes et les structures de données du programme. Il permet aussi d'exprimer des structures de parallélisme plus complexes et arbitraires, comme le parallélisme hiérarchique, sans avoir à les gérer explicitement comme dans les langages partitionnés, offrant ainsi un niveau de granularité arbitraire du parallélisme. Par contre, ces langages sont beaucoup plus difficiles à implanter, car le compilateur et les librairies de soutien doivent gérer le partitionnement et la distribution des données ainsi que le balancement de charge tout en tentant d'offrir de bonnes performances.

Nous allons maintenant présenter les langages orientés vers le parallélisme qui sont, à notre avis, toujours importants et d'actualité pour le traitement parallèle en mémoire partagée et distribuée.

2.3.1 Multilisp

Multilisp [15], un dialecte du langage fonctionnel *Scheme* [1], fournit un modèle *global* de parallélisme. Il permet d'indiquer les sections de code pouvant être exécutées

en parallèle grâce à une simple primitive qui est utilisable de manière hiérarchique. Les zones ainsi identifiées représentent du parallélisme potentiel qui ne sera utilisé que si des ressources sont libres dans le système. En effet, ce dernier se charge de partager les tâches entre les processeurs de la machine grâce à la technique du *vol de tâches* où un processeur au repos tente de soutirer une tâche exécutable disponible sur un autre processeur.

La *Création paresseuse de tâches* est une approche à l'implantation du vol de tâches de *Multilisp*. Elle a été développée par Feeley [13] et Mohr [20]. Dans cette dernière, le système conserve des pointeurs vers la pile d'exécution du programme pour indiquer les tâches qui peuvent être volées par les autres processeurs. Une tâche n'est réellement créée à partir de ce pointeur que lorsqu'un vol est tenté.

Multilisp est bien approprié pour exprimer du parallélisme de tâche hiérarchique à granularité arbitraire, mais il n'offre pas de manière simple d'accomplir du parallélisme de données, par exemple du travail sur des vecteurs distribués. De plus, Multilisp a historiquement été limité aux machines à mémoire partagée et ne fonctionne pas dans un contexte de mémoire distribuée.

2.3.2 Cilk

Cilk [5] est un dialecte du langage *C* largement répandu qui offre des primitives pour le parallélisme semblable à ce qu'on retrouve dans *Multilisp*. Comme *Multilisp*, il offre un modèle global de parallélisme où la responsabilité d'identifier les zones parallélisable incombe au programmeur. Le travail est distribué par *vol de tâche* et le système implante la *création paresseuse de tâche*. Le parallélisme peut être hiérarchique et la granularité est donc arbitraire.

Cilk souffre des mêmes faiblesses que *Multilisp*. Il ne fonctionne que sur les machines à mémoire partagée et ne permet pas de parallélisme sur des structures de données distribuées. Il est toutefois très utilisé et depuis septembre 2010, *Intel* fourni sa propre version de *Cilk*, *Cilk Plus*, destinée aux programmeurs développant sur les plateformes

multicoeurs.

2.3.3 MPI

Dans le domaine des langages destinés aux machines distribuées, la librairie *MPI* [21] reste dominante. Comme nous l'avons vu, c'est une librairie de passage de message où le programmeur doit effectuer l'ensemble de la distribution de tâches et de données manuellement. Cette librairie offre un modèle *programme unique données multiples*. En effet, un certain nombre de copies du programme sont exécutées, en général autant que le nombre de processeurs, et la granularité du parallélisme est donc limitée au nombre de processus. Ce modèle s'appuie sur des primitives de communication un à un et collectives ainsi que des opérations de synchronisation des processus, barrières et autres.

Le programmeur doit décomposer manuellement son programme en tâches statiques et diviser les structures de données en morceaux qu'il devra distribuer aux processus à l'aide des riches primitives de passage de messages que la librairie offre. *MPI* offre un très bon niveau de performance et est massivement utilisé, mais souffre énormément de tous les problèmes inhérents aux langages parallèles partitionnés ainsi que de ceux des langages de bas niveau.

2.3.4 CAF, UPC et Titanium

Co-Array Fortran [23], *Unified Parallel C* [11] et *Titanium* [28], sont des langages offrant le modèle *Programme unique données multiples* et y ajoutant le concept appelé *espace d'adressage global partitionné* qui est une mémoire partagée distribuée partitionnée entre les différents processus formant le système. Chaque processus peut écrire et lire sur cette mémoire et elle est habituellement implantée en utilisant des librairies de communication spécialisées de bas niveau. Par rapport à *MPI*, cet espace d'adressage permet de consulter des emplacements mémoires et des variables dans les autres processus et d'offrir certaines abstractions concernant les structures de données distribuées.

Tous ces langages offrent des opérations de synchronisation telles que les barrières.

1. **CAF** : Dialecte de *Fortran* qui offre le concept de *co-array*. Un *co-array* est une variable existant sur tous les processus formant le système et qui peut être référencée à distance à l'aide d'un indice non standard en *Fortran*. Cette abstraction permet d'éviter les échanges de messages pour le partage des structures de données.

2. **UPC** : Dialecte de *C*, qui offre aussi un concept de variable partagée d'une manière différente de *CAF*. En *UPC*, une variable *shared* sera présente dans un seul processus, mais pourra être consultée par tous. Un vecteur *shared* sera quant à lui distribué automatiquement entre les processus selon une taille de bloc définie par l'usager lors de la déclaration du vecteur. *UPC* offre aussi une primitive de partage de travail équivalant à une boucle `for`, mais qui permet de préciser comment les itérations de la boucle seront distribuées entre les processus.

3. **Titanium** : Dialecte de *Java* fournissant l'espace mémoire partitionné en plus d'opérations de communication semblables à celles de *MPI*. Il est possible de définir des structures de données distribuées comme dans *UPC* et le même genre de primitives de synchronisation existe. Étant basé sur *Java*, Titanium a l'avantage d'être un langage offrant beaucoup plus d'abstractions que *CAF* et *UPC* en plus de bénéficier d'un *ramasse-miette*.

2.3.5 Chapel, X10 et Fortress

Chapel [7, 8, 17], *Fortress* [19, 25] et *X10* [10, 18] ont été conçus dans le cadre du projet *High Productivity Computer Systems HPCS* de *DARPA* qui vise à financer le développement de système à productivité élevée en informatique de haute-performance. Ces langages partagent tous comme objectif de faciliter le développement d'applications parallèles destinées à la nouvelle génération d'ordinateurs parallèles, les grappes non uniformes de noeuds multicoeurs [27]. Dans cette optique, les concepteurs de ces

langages considèrent que le modèle fragmenté actuel est néfaste pour le développement et réduit la productivité. C'est pourquoi ils offrent un modèle global ainsi qu'un espace d'adressage global partitionné. Ils supportent tous le parallélisme de tâches hiérarchique explicite et implicite à l'aide d'éléments du langage comme des boucles parallèles. Enfin, ils fournissent le parallélisme de données sur les structures distribuées.

Fortress est développé par *Sun Microsystems* comme un langage visant les grands systèmes parallèles. En plus des éléments mentionnés plus haut, les développeurs de *Fortress* ont voulu faire du langage un outil adapté à la communauté scientifique qui utilise l'informatique de haute performance. C'est pourquoi la syntaxe de celui-ci se rapproche beaucoup de la notation mathématique. Afin d'appuyer cet objectif, le langage est aussi statiquement typé et supporte la vérification des types d'unités physiques dans les équations. *Fortress* offre aussi le concept de *transaction* qui permet d'effectuer des opérations de manière atomique sur des zones mémoire partagées [16]. Le parallélisme de tâches est supporté par des boucles implicitement parallèles, l'évaluation parallèle des paramètres de fonctions ainsi que l'introduction explicite de parallélisme. Toutes ces constructions indiquent du parallélisme potentiel qui est partagé à l'aide d'une séparation diviser pour régner et du vol de tâches qui s'inspirent de [3]. *Fortress* devrait supporter le traitement sur les machines à mémoire distribuée à l'aide du concept de *régions* représentant la structure de la machine. Ces *régions* devraient pouvoir être décrites de manière hiérarchique permettant d'exprimer des coûts de communications différents entre des ensembles de machines et, au plus bas niveau, consister d'un ensemble d'unités de calcul partageant une mémoire.

X10 est un dialecte de *Java* développé par *IBM* et tente de réutiliser une grande quantité de concepts de ce dernier au lieu de développer un nouveau langage. Toutefois, il remplace tous les concepts de parallélisme propre à *Java* par ses propres primitives. Pour la distribution des données et du travail, *X10* introduit le concept de *places* qui représentent directement un groupe de cpus partageant une mémoire locale. Le langage supporte aussi le parallélisme de tâche à l'aide d'une primitive d'exécution parallèle

explicite permettant d'exécuter une expression à une *place* particulière ainsi qu'une façon d'exécuter du code de manière atomique.

Enfin, *Chapel* est un nouveau langage développé par *Cray* qui emprunte des concepts d'un grand nombre de langages. Le parallélisme de tâches ressemble plus à celui de *Fortress* que celui de *X10*. L'expression du parallélisme se fait de manière plus implicite à l'aide de boucles et autres primitives du langage. Chapel utilise aussi le vol de tâches comme technique de balancement de charge.

2.4 Multilisp

Il existe tout un spectre de manières d'introduire le parallélisme dans un programme. D'un côté, on trouve des langages où le parallélisme est implicite, c'est-à-dire que le programmeur n'a pas à manipuler explicitement des opérateurs ou fonctionnalités l'introduisant. À l'autre bout se trouvent les langages où le parallélisme est introduit directement par le programmeur à l'aide de mécanismes du langage.

Multilisp [15] est un langage de programmation fonctionnel dérivé de Scheme [1] auquel sont ajoutées des manières d'introduire explicitement le parallélisme dans un programme. Multilisp a été conçu pour être utilisé sur des machines à mémoire partagée. En effet, il permet les changements d'état (affectations) sur la mémoire. Il libère aussi le programmeur de la tâche d'effectuer le partage des tâches, le *partitionnement*, car le système s'en charge automatiquement.

Le modèle de parallélisme de Multilisp est une façon simple et efficace de programmer des algorithmes parallèles. Il permet au programmeur de déclarer les zones de code pouvant être exécutées de manière concurrente. Ce modèle permet au développeur de prendre des décisions et de porter des jugements quant au coût-bénéfice d'introduire du parallélisme qu'un compilateur peut parfois difficilement découvrir en examinant le programme. De plus, il permet aussi d'utiliser Multilisp comme langage intermédiaire d'un langage parallèle plus abstrait.

Dans cette section, nous présentons tout d'abord le modèle de programmation de

Multilisp. Nous expliquons comment les opérateurs sont utilisés pour introduire le parallélisme et comment Multilisp peut s'adapter à différents types de calculs parallèles. Nous expliquons ensuite le concept de partitionnement et du même coup un des points importants de Multilisp, son usage du partitionnement dynamique. Nous présentons plusieurs exemples de programmes parallèles où le partitionnement dynamique apporte un avantage certain.

2.4.1 Continuations

Avant de se plonger dans l'étude de Multilisp, il est nécessaire d'aborder le sujet des *continuations*, car ces dernières ont un rôle important dans la conceptualisation de ce langage. Nous ne donnons ici qu'un bref aperçu du sujet. Il est toutefois traité beaucoup plus en détail dans le chapitre 3.

Une *continuation* est une abstraction représentant le reste de l'exécution du programme à un point donné dans son exécution. La continuation d'un appel de fonction contient tout ce qui sera fait après que cette dernière ait retourné. Si on prend comme exemple la fonction `append` :

```
(define (append lst1 lst2)
  (if (pair? lst1)
      (let ((head (car lst1))
            (tail (cdr lst1)))
        (cons head (append tail lst2)))
      lst2))
```

```
> (print (append '(1 2) '(3 4)))
>> (print (cons 1 (append '(2) '(3 4))))
```

L'appel à `(append '(1 2) '(3 4))` peut être réduit à un appel à `(cons 1 (append '(2) '(3 4)))`. La continuation de l'appel `(append '(2) '(3 4))` sera alors l'appel à `(cons 1 <resultat>)` suivit de l'appel à `print`. Il est possible, sachant cela, de représenter cette continuation par une fonction effectuant les mêmes opérations.

```
(lambda (<resultat>)
  (print (cons 1 <resultat>)))
```

2.4.2 Opérateurs Multilisp

En Multilisp, la possibilité d'évaluation parallèle d'une expression est indiquée par l'opérateur `future`. Lorsqu'une expression est contenue dans un `future`, elle peut être évaluée en même temps que sa continuation. Toutefois, les *opérations strictes* doivent bloquer en attendant le résultat du `future`. Une *opération stricte* est une opération dont certains des opérandes doivent être *déterminés* avant qu'elle puisse commencer son travail. Un opérande est *déterminé* lorsque l'opération calculant cette dernière a terminé et rendu disponible son résultat. Par exemple, dans la fonction fibonacci :

```
(define (fib n)
  (if (< n 2)
      n
      (let ((r1 (future (fib (- n 1)))))
        (+ r1 (fib (- n 2))))))
```

l'évaluation de `(fib (- n 1))` peut être faite de manière concurrente avec l'évaluation de `(fib (- n 2))` qui est dans la continuation de `(future (fib (- n 1)))`. Par contre, l'addition étant une opération stricte sur tous ses opérandes, incluant `r1` dans ce cas, elle ne pourra s'accomplir que lorsque `(fib (- n 1))` et `(fib (- n 2))` auront terminé.

Une autre facette de Multilisp est qu'il supporte les effets de bord. Cela rend le langage indéterminé. Un exemple de ce comportement est l'expression

```
(let ((x 0))
  (future (set! x 1))
  x)
```

Cette expression pourrait autant retourner la valeur `0` que la valeur `1`. Pour cette raison, un autre opérateur existe en Multilisp. Il s'agit de `touch` qui peut être utilisé pour forcer des contraintes d'ordonnancement d'effets de bord. L'opérateur `touch` est en fait une opération d'identité stricte ce qui force l'attente du résultat calculé dans un `future`.

2.4.3 Types de parallélisme en Multilisp

Le style de programmation de Multilisp se prête naturellement à la conception de programme utilisant le parallélisme de tâches. En effet, l'opérateur `future` permet de facilement diviser les différentes parties d'un programme en sous-tâches à accomplir. Par exemple dans le cas simple où 3 tâches différentes doivent être exécutées sur une même donnée

```
(define (program data)
  (let* ((r1 (future (task1 data)))
         (r2 (future (task2 data)))
         (r3 (task3 data)))
    ...
    (touch r1)
    ...
    (touch r2)
    ...))
```

Une autre façon d'introduire ce type de parallélisme est une variante qu'on appelle le *parallélisme de pipeline* où la production d'une donnée est faite de manière concurrente à sa consommation. Un exemple est la construction d'une liste et l'utilisation des parties déjà construites dans la procédure `pmap`

```
(define (pmap proc lst)
  (if (pair? lst)
      (let ((tail (future (pmap proc (cdr lst)))))
        (let ((val (proc (car lst))))
          (cons val tail)))
      '()))
```

Comme l'opérateur `cons` n'est pas un opérateur strict, la paire est retournée immédiatement et la queue de la liste se trouve à être calculée de manière concurrente avec la continuation de `pmap`. La synchronisation se fait lorsque la continuation doit accéder à la queue de la liste. À ce moment, la tâche peut être suspendue si le résultat n'est pas prêt.

En étudiant la procédure `pmap`, on réalise qu'il ne s'agit pas uniquement de parallélisme de tâche. Il est vrai que deux parties de l'algorithme se chevauchent, soit la

production et la consommation de la liste. Mais comme la quantité de parallélisme varie en fonction de la taille de la liste, il s'agit aussi d'une forme de parallélisme de données.

De plus, un autre point intéressant de `pmap` est qu'elle *exporte le parallélisme* ce qui signifie que du travail qui est démarré à l'intérieur de la procédure pourra encore être en exécution lorsque cette dernière aura terminé. Cela peut devenir problématique lorsque des effets de bord sont présents dans le travail qu'on effectue de manière concurrente. Si, par exemple, la procédure appliquée par `pmap` modifie un état global, il n'y a aucune garantie que les modifications auront toutes été complétées dans la continuation de `pmap`. Voici la procédure `pmap` modifiée à l'aide de l'opérateur `touch` pour éliminer cette possibilité :

```
(define (pmap proc lst)
  (if (pair? lst)
      (let ((tail (future (pmap proc (cdr lst)))))
        (let ((val (proc (car lst))))
          (cons val (touch tail))))
      '()))
```

Ici, on force la synchronisation de la queue de la liste avant le retour de `pmap` afin de s'assurer que l'ensemble du travail qui doit être fait dans la queue de la liste a bien été complété. Cela peut potentiellement suspendre la continuation du `future`. Bien entendu, on vient de limiter les possibilités de parallélisme, car le chevauchement de la production et de la consommation n'est plus possible dans cette version. C'est pourquoi si on sait que la procédure qu'on applique dans `pmap` ne cause pas d'effet de bord devant être exécutés dans un certain ordre, on aurait avantage à utiliser la première version.

La grande limitation de la procédure `pmap` est qu'elle travaille sur une structure de donnée fondamentalement séquentielle et qui passe donc mal à l'échelle. Il serait préférable d'utiliser des structures qui passent mieux à l'échelle comme les vecteurs ou les arbres. Cela permet de démarrer les `futures` en utilisant un algorithme *diviser pour régner* qui permettra de diviser le fil d'exécution en deux tâches mieux balancées qu'un démarrage séquentiel. Voici la méthode `pvmap!` qui applique une fonction sur chacun des éléments d'un vecteur et effectue une mutation sur ce dernier pour entreposer le

nouvel élement

```
(define (pvmap! proc vect)
  (define (map-range! proc lo hi)
    (if (= lo hi)
        (vector-set! vect lo (proc (vector-ref vect lo)))
        (let ((mid (quotient (+ lo hi) 2)))
          (let ((sync (future (map-range! proc (+ mid 1) hi))))
            (map-range! proc lo mid)
            (touch sync)))))
  (map-range! proc 0 (- (vector-length vect) 1))
  vect)
```

Comme `pvmap!` effectue une synchronisation forcée, tous les effets de bord auront terminé lorsqu'elle retournera. Le parallélisme *diviser pour régner* est un idiome très naturel et qui s'applique très bien aux algorithmes récursifs et travaillant sur des arbres. C'est la manière préférable d'exprimer le parallélisme en Multilisp.

2.4.4 Partitionnement

Le problème fondamental du parallélisme est la distribution du travail aux unités de calcul afin de maximiser leur efficacité, c'est-à-dire de maximiser la quantité de travail utile qu'elles accompliront et de minimiser le travail de gestion. Un préalable à ce problème est la division de ce qui doit être accompli en sous-tâches pouvant être distribuées. C'est le problème du partitionnement. Il existe deux méthodes, le partitionnement statique et le partitionnement dynamique.

Le partitionnement statique consiste à diviser, avant la compilation, le travail en sections de taille prédéterminées. Lorsque la donnée sur laquelle le programme travaillera est connue d'avance et que la structure de l'algorithme n'est pas trop complexe, cette approche peut s'avérer utile, mais elle s'adapte mal aux données de taille variable et à des programmes ayant une grande complexité.

En effet, lorsque la taille des données peut varier, le partitionnement ne peut être fait de manière statique, car avec une donnée plus grande, il est possible de créer plus de tâches. De plus, lorsque le programme est complexe, et qu'il est composé de plusieurs modules, il existe alors plusieurs niveaux de parallélisme. Il pourra y avoir

du parallélisme à l'intérieur des modules, du parallélisme entre modules séquentiels et même entre modules parallèles. Un module parallèle qui est réutilisé plusieurs fois dans le programme pourra parfois s'exécuter de manière indépendante.

Par exemple, sur un système à n processeurs, un module parallèle à grain très fin pourrait rouler seul. On devrait alors effectuer un partitionnement pour obtenir n tâches. Plus tard, le même module pourrait être invoqué de nouveau avec des paramètres différents et être exécuté de manière concurrente avec *n-1* autres modules séquentiels. On aurait alors besoin de procéder à aucun partitionnement. C'est cette possibilité de s'adapter à des structures de parallélisme complexes ainsi que d'utiliser des informations seulement disponibles à l'exécution comme le nombre de processeurs sans travail qui rend le partitionnement dynamique attrayant.

Cette versatilité possède un coût. Le partitionnement entraine des dépenses de temps machine et d'espace pour la création des tâches, le maintien de leur état et la prise de décisions quant au partitionnement. Le défi de l'implantation d'un tel système est de trouver un bon compromis entre le parallélisme et les dépenses supplémentaires.

On pourrait considérer que le problème du partitionnement en Multilisp est extrêmement simple, car chaque `future` a pour effet de créer une tâche. Toutefois, il existe d'autres décisions d'implantation que le simple fait de créer ou non des tâches. Par exemple, l'implantation qui est présentée plus loin dans ce document, la *création paresseuse de tâches* [20], base sa stratégie de partitionnement dynamique sur la possibilité de plusieurs représentations différentes des tâches avec des coûts et des particularités différentes.

2.5 Passage de messages

Le passage de message est un paradigme de programmation concurrente utilisé dans les systèmes parallèles à mémoire distribuée. Dans ce modèle, chaque processus possède une mémoire locale privée et communique avec les autres processus au moyen de l'envoi et de la réception de messages.

Ce modèle est beaucoup utilisé aujourd'hui à cause de la prolifération des systèmes parallèles de type *cluster* composés d'une grande quantité de machines relativement peu puissantes reliées en réseau local. Ces machines ont peu à peu remplacé les gros multiprocesseurs à mémoire partagée à cause des coûts beaucoup moins élevés pour des performances comparables.

Deux principaux systèmes de passage de messages existent. Il s'agit du *Message Passing Interface* [21] qui est une librairie en C proposant un ensemble de primitives de communication et de synchronisation permettant la coopération de plusieurs processus indépendants sur une tâche. L'autre système est *Erlang* [26] qui est un langage de programmation fonctionnelle concurrente qui permet la création de processus légers sans états partagés et communiquant par le passage de message asynchrone.

Dans cette section nous présenterons à la fois *MPI* et *Termite*, une variante de Scheme inspirée d'Erlang et implantée sur le compilateur *Gambit-C*. Pour chacun des systèmes, nous décrirons le modèle, présenterons ses opérations, et montrerons des exemples de programmes.

2.5.1 Termite

Termite [14] est un langage de programmation fonctionnelle concurrente dérivé de Scheme et dont les primitives de concurrence sont inspirées d'Erlang. Il offre des interfaces de haut niveau simples et robustes qui permettent de se concentrer sur le développement de protocoles distribués en ignorant les détails de bas niveau.

Par rapport à Erlang, on remarque deux additions importantes : les macros et les continuations pouvant être envoyées dans les messages comme n'importe quel autre objet Scheme, ce qui permet l'implantation simple de concepts tels que la migration de tâches et la mise à jour dynamique de code.

Nous présentons ce système, car il est utilisé comme plateforme pour la réalisation de notre système de passage de messages ainsi que pour notre système de partitionnement dynamique. L'utilisation d'un tel système est justifiée par le fait de pouvoir ignorer les

détails tels que l'ouverture et le maintient des canaux de communication, l'encodage et le décodage de nos protocoles ainsi que la possibilité d'erreurs et enfin, le fait de pouvoir utiliser plusieurs processus indépendants sans avoir à se soucier de l'acheminement des messages. Enfin, la possibilité d'inclure des continuations dans les messages envoyés était essentielle pour le développement du système de partitionnement dynamique.

2.5.1.1 Processus

Les processus Termite sont des processus légers indépendants les uns des autres. Un processus ne peut pas accéder ni modifier la mémoire d'un autre processus. Cela enlève le besoin de modéliser l'exclusion mutuelle et est beaucoup plus en accord avec la réalité des systèmes distribués où les processus sont sur des machines différentes.

Ils sont implantés à l'aide des threads du compilateur *Gambit-C* qui sont très peu coûteux. Il est pensable d'en créer des millions sur un système sans perte de performance notable.

Chaque processus s'exécute sur un *noeud* qui est en fait un processus lourd du système d'exploitation exécutant Termite. Ils possèdent tous un identifieur unique dans le système qui permet à n'importe quel processus Termite situé sur un noeud quelconque d'envoyer un message à un processus distant dont il connait l'identificateur. On appelle cet identifiant le *pid* et l'implantation de leur création garantit qu'ils sont uniques de manière globale.

2.5.1.2 Opérations

Chaque processus Termite possède une *boîte aux lettres* qui entrepose les messages dans l'ordre où ils sont reçus. L'opérateur de base pour la réception de message est `?`. Il permet au processus de bloquer jusqu'à ce qu'un message soit disponible. L'envoi de message se fait à l'aide de l'opérateur `!` qui envoie un message à un processus arbitraire. C'est un opérateur asynchrone, c'est-à-dire qu'il ne bloquera pas et les messages peuvent être un sous-ensemble des types d'objets Scheme qui sera décrit plus loin. Finalement,

la création de processus se fait à l'aide de la procédure `spawn` qui démarre un processus exécutant la fonction qui lui est passée en paramètre. Voici un exemple simple de communication entre deux processus créés sur le même noeud :

```
(define p1
  (spawn
    (lambda ()
      (display (?)))))

(define p2
  (spawn
    (lambda ()
      (! p1 (list 'Hello 'world)))))

=> (Hello world)
```

Les objets pouvant être transmis comme message sont limités aux objets Scheme sérialisables, incluant les continuations et les processus pour lesquels on envoie leur identificateur. Cela exclut notamment les objets reliés à la synchronisation des threads comme les mutex et les variables de condition ainsi que les objets attachés à des périphériques systèmes tels que les ports de communication, les fichiers, les canaux réseau. Toutefois, ces derniers peuvent être rendus disponibles si on les encapsule derrière des processus qu'on appelle *proxy* et qui permettent de présenter une interface à ces ressources pour l'ensemble des processus présents dans le système distribué.

L'opérateur `?` peut être utilisé de manière plus complexe. En effet il possède deux paramètres optionnels, un temps d'attente maximal et une valeur à retourner par défaut. On peut donc l'utiliser de manière asynchrone. De plus, la réception sélective de messages est aussi possible avec l'opérateur `??` auquel on fournit un prédicat et qui retourne le premier message le satisfaisant.

```
(? [timeout [default]])
(?? predicate [timeout [default]])
```

Enfin pour faciliter la réception sélective et la *déstructuration* des messages, Termite fournit l'opérateur `recv`. Il permet de spécifier des conditions sur la structure des messages de manière déclarative, d'extraire des parties du message et de les entreposer dans

des variables arbitraires ainsi que de filtrer sur des prédicats généraux. On peut implanter, à l'aide de `recv` l'opérateur `!?` qui envoie un message à un processus et attend une réponse. On utilise la procédure `make-tag` qui permet de créer un identificateur unique dans le système.

```
(define (!? process message)
  (let ((tag (make-tag)))
    (! process (list (self) tag message))
    (recv
      ((reply-tag reply-message)
       (where (equal? reply-tag tag))
       reply-message))))
```

Enfin, on peut utiliser toutes ces opérations pour définir un processus qui répond à une demande de *ping*.

```
(define (pong-server)
  (recv
    ((from tag 'ping)
     (! from (list  tag 'pong))
     (pong-server))))

(let ((pong (spawn pong-server)))
  (print (!? pong 'ping)))

=> pong
```

Il est aussi possible de démarrer un processus sur un noeud distant avec l'opération `remote-spawn`. À l'aide de la sérialisation des continuations, il est alors possible par exemple d'implanter un opérateur de migration permettant à un processus de s'exiler sur un autre noeud.

```
(define (migrate-process node)
  (continuation-capture
    (lambda (k)
      (remote-spawn node (lambda ()
                           (continuation-return k #t)))
      (halt!))))
```

2.5.2 Message Passing Interface

Le *Message Passing Interface* est une interface de communication permettant le passage de message et la synchronisation entre un ensemble de processus pouvant se trouver sur des noeuds distants. Il permet la communication directe entre deux noeuds ainsi que les communications collectives de type *broadcast*, *scatter* et *gather*. MPI est aujourd'hui le standard *de facto* pour le calcul scientifique haute performance et la programmation des ordinateurs massivement parallèles.

Nous présentons ce système, car il est l'inspiration principale pour notre système de passage de message. Nous tentons d'offrir un sous-ensemble des fonctionnalités de MPI permettant de créer des applications distribuées relativement complexes.

2.5.2.1 Processus

Les processus MPI sont des processus lourds du système d'exploitation qui possèdent chacun leur mémoire privée que les autres ne peuvent accéder. Dans un cas typique d'utilisation, un seul processus MPI est créé par processeur présent dans le système distribué. Les processus sont regroupés en ensembles appelés *communicators* qui permettent d'effectuer des communications collectives à l'intérieur et entre différents groupes de processus. Ils sont créés à partir d'un noeud à l'aide de la commande *mpirun* qui se charge de démarrer le nombre demandé sur les noeuds spécifiés. Par exemple si on a un programme appelé `hello-world` et qu'on veuille rouler 2 processus sur le noeud `foo` et deux sur le noeud `bar`.

```
mpirun -np 4 --host foo,bar hello-world
```

2.5.2.2 Opérations

MPI possède une librairie de fonctionnalité extrêmement riche. Il existe des opérations de communication, de synchronisation, de gestion dynamique des processus, etc. Nous ne couvrons qu'un petit sous-ensemble dans cette section.

Tout d'abord, chaque processus MPI possède un identificateur numérique unique dans son communicator. Ce dernier est appelé le *rang*. Les procédures `MPI_Comm_rank` et `MPI_Comm_size` permettent respectivement à un processus de connaitre son rang et la taille de son communicator. Voici le programme `hello-world` utilisé dans l'exemple précédent.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

  int numprocs;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Process %i of %i says : Hello, World!\n", rank, numprocs);

  MPI_Finalize();

}
```

Dans MPI les communications de base se font à l'aide des opérations point à point. Elles permettent à un processus d'envoyer un message à un autre processus. Ces opérateurs de base sont synchrones, un processus qui envoie un message va bloquer tant qu'il ne recevra pas une confirmation de la réception. Ces échanges de messages se font à l'aide des procédures `MPI_Send` et `MPI_Receive`.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Voilà un autre exemple de `hello-world`, cette fois avec le processus 0 qui envoie le message au processus 1.

```
#include <mpi.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {

  int rank;

  char *message;
  int size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if( rank == 0 ) {
    message = "Hello, World!";
    size = strlen(message) + 1;
    MPI_Send(&size, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Send(message, size, MPI_CHAR, 1, 0, MPI_COMM_WORLD);

  }
  else if( rank == 1 ) {
    MPI_Recv(&size
             , 1
             , MPI_INT
             , 0
             , 0
             , MPI_COMM_WORLD
             , MPI_STATUS_IGNORE);
    message = malloc( size * sizeof(char) );
    MPI_Recv( message
              , size
              , MPI_CHAR
              , 0
              , 0
              , MPI_COMM_WORLD
              , MPI_STATUS_IGNORE);
    printf("Message received on node %i : %s\n"
           , rank
           , message);
  }

  MPI_Finalize();

}
```

Les opérations de communication collective permettent d'abstraire des *patterns*

d'échange de données plus complexes. Elles sont invoquées de manière uniforme, c'est-à-dire que les processus envoyant les données feront le même appel de procédure que ceux les recevant. Par exemple, l'opérateur `MPI_Bcast` permet à un processus d'envoyer un message à tous les autres processus d'un communicateur donné.

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
            int root, MPI_Comm comm)
```

Dans cet exemple, le processus 0 envoie le vecteur `{1, 2, 3}` à tous les autres processus présents dans le communicateur `WORLD`. Le quatrième paramètre, nommé `root`, indique l'origine du message, dans ce cas `0`.

```
int *v;

if( rank == 0 )
  v = {1, 2, 3}
else
  v = malloc(3 * sizeof(int));

MPI_Bcast( v, 3, MPI_INT, 0, MPI_COMM_WORLD );
```

Enfin, l'opérateur `MPI_Scatter` permet à un processus d'envoyer un sous-ensemble de taille égale d'une donnée à chacun des autres processus et `MPI_Gather` de recevoir une donnée de même taille de chaque processus et de les entreposer en ordre.

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
            void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
            MPI_Comm comm)

int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
            void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
            MPI_Comm comm)
```

Voici un exemple où le processus `0` distribue un élément d'un vecteur à chacun des autres processus et chacun d'entre eux multiplie l'élément par son rang.

```
int *v;
int elem;

if( rank == 0 )
```

```
  v = malloc(numprocs * sizeof(int))

MPI_Scatter(v,1,MPI_INT,&elem,1,MPI_INT,0,MPI_COMM_WORLD);
elem *= rank;
MPI_Gather(&elem,1,MPI_INT,v,1,MPI_INT,0,MPI_COMM_WORLD);
```

CHAPITRE 3

IMPLANTATION DES CONTINUATIONS

Les *continuations* jouent un rôle fondamental dans le modèle d'exécution de tous les langages de programmation. En effet, la gestion des appels de fonctions en dépend. Puisqu'une continuation représente l'état des calculs suspendus suite à une chaîne d'appels de fonctions, elles sont intimement reliées au concept de processus et du même coup au concept de *future*. Une tâche pouvant être essentiellement considérée comme une *continuation*, le déplacement de tâches doit reposer sur la *capture de continuations* et son efficacité est donc primordiale.

Dans ce chapitre, nous allons tout d'abord introduire les *continuations*. Nous expliquerons les concepts abstraits, présenterons des exemples d'usage des continuations ainsi que des possibilités d'implantations simples. Par la suite, nous décrirons plusieurs implantations utilisées dans divers compilateurs existants. Enfin, nous expliquerons la stratégie d'implantation utilisée dans *Gambit-C* et nous donnerons des détails sur chacune des facettes de celle-ci. Nous aborderons l'appel et le retour de fonction, la capture de continuation, le changement de contexte ainsi que les détails concernant la représentation des *frames*. Finalement, nous discuterons de la performance de l'implantation choisie et nous conclurons.

3.1 Concepts des continuations

Une *continuation* est une représentation abstraite de l'état du contrôle qui est souvent concrétisée par une chaîne de *frames* appelés *frames de continuation*. [9] Dans le langage *Scheme*, elle peut être capturée, c'est-à-dire encapsulée dans un objet de première classe, pour pouvoir être invoquée de manière répétée. L'invocation d'une continuation capturée signifie le retour au point de contrôle qui existait au moment de sa capture.

Dans la chaîne de *frames*, chaque *frame* représente un appel de fonction non terminé.

```
(define (find pred? lst)
  (call/cc
    (lambda (return)
      (for-each (lambda (el)
                  (if (pred? el)
                      (return el)))
                lst)
      #f)))
```

Figure 3.1 – Exemple de sortie non locale avec continuations

Le *frame* contient l'information nécessaire à la complétion de cet appel, c'est-à-dire les variables requises ainsi que le point de retour dans le code exécutable. Lorsque l'appel retourne, le contrôle passe alors à la *continuation parente* qui est la chaîne de *frames* dont on a coupé le premier *frame*. La *continuation parente* reçoit alors la valeur de retour de l'appel de fonction. Le *frame* initial (le plus profond) de la *continuation* indique la fin du programme. [13]

3.1.1 Exemples d'usage de continuations

Dans le langage *Scheme*, une continuation peut être capturée à l'aide de l'opérateur `call-with-current-continuation` souvent abrévié à `call/cc`. Cet opérateur se charge de capturer le point de contrôle courant et de l'encapsuler dans une procédure prenant un paramètre. Lors de l'appel de cette procédure, la continuation capturée sera remise en place comme continuation courante et le paramètre lui sera communiqué comme valeur de retour.

Pour illustrer le concept, à la figure 3.1, on peut voir un exemple d'usage de continuations pour effectuer une sortie non locale. On y définit une procédure `find` qui reçoit comme paramètres un prédicat et une liste. `Find` capture sa continuation et parcourt ensuite la liste à l'aide de l'opérateur `for-each`. Lorsqu'elle rencontre le premier élément respectant le prédicat, elle invoque la continuation capturée avec ce dernier, ce qui a pour effet de remettre en place la chaîne de *frames* qui existait lors de l'entrée

```
(define fail
  (lambda ()
    (error "can't backtrack")))

(define (in-range a b)
  (call/cc
   (lambda (cont)
     (enumerate a b cont))))

(define (enumerate a b cont)
  (if (> a b)
      (fail)
      (let ((save fail))
        (set! fail
          (lambda ()
            (set! fail save)
            (enumerate (+ a 1) b cont)))
        (cont a))))

(let* ((x (in-range 1 9))
       (y (in-range 1 9))
       (z (in-range 1 9)))
  (if (= (* x x)
         (+ (* y y) (* z z)))
      (write (list x y z))
      (fail)))

=> (5 3 4)
```

Figure 3.2 – Exemple de retour arrière avec continuations

dans la procédure et de lui passer comme résultat l'élément trouvé. `find` retournera alors à son appelant avec cet élément comme valeur de retour.

Un exemple plus complexe d'usage des continuations est présenté à la figure 3.2. Dans ce programme, les continuations ne sont plus utilisées une seule fois comme dans l'exemple de la sortie non locale, mais peuvent être invoquées plus d'une fois.

Ce programme tente de trouver la première combinaison de trois nombres a, b, c respectant l'identité $a^2 = b^2 + c^2$. La première procédure, `in-range`, a simplement pour tâche de capturer une continuation. On voit donc que les continuations qui seront capturées correspondront aux points de retour des trois appels à `in-range`. La partie essentielle du travail est effectuée par la procédure `enumerate`. Elle reçoit en paramètres un premier nombre à essayer, un deuxième nombre qui est une limite et une continuation. Elle suppose aussi l'existence d'une fonction de retour arrière nommée `fail`. Chaque appel à `enumerate`, si le paramètre `a` ne dépasse pas la limite `b`, installe une nouvelle fermeture de retour arrière qui a pour travail, lorsqu'elle est invoquée, de rétablir l'ancienne et d'appeler de nouveau `enumerate` avec le nombre suivant. L'utilité des continuations, dans ce programme, est d'indiquer à `enumerate` à quel point de retour le résultat doit être envoyé. Si le nombre à essayer est plus grand que la limite, la fermeture de retour arrière sera immédiatement appelée et il s'agit soit d'une fermeture installée par un appel précédent à `in-range` ou bien de la procédure d'erreur initiale.

On peut donc constater que, lors de l'exécution du code, les trois premiers appels à `in-range` retourneront tous `1` comme résultats et, l'identité n'étant pas respectée, la procédure de retour arrière installée par le dernier appel à `enumerate`, celle correspondant à la liaison de la variable `z`, sera invoquée. Elle remettra en place la procédure de retour arrière correspondant à la variable `y` et appellera `enumerate` avec la valeur `2` et la continuation correspondant à la variable `z`. L'identité ne pouvant être respectée avec aucune valeur assignée à la variable `z` si les variables `x` et `y` valent `1`, `enumerate` sera éventuellement appelée avec la valeur `10`. À ce moment, la procédure de retour arrière installée à l'intérieur de l'appel à `in-range` liant la variable `y` sera appelé. Cette

```
(define (eval expr)
  (if (number? expr)
      expr
      (case (car expr)
        ((+)
         (let ((n1 (eval (cadr expr)))
               (n2 (eval (caddr expr))))
           (+ n1 n2)))
        ((-)
         (let ((n1 (eval (cadr expr))))
           (- n1))))))

(eval '(- (+ 5 4)))

=> -9
```

Figure 3.3 – Simple évaluateur pour les exemples de continuations

dernière invoquera `enumerate` avec le paramètre `a` à `2` et la continuation correspondant à la liaison de la variable `y`. On aura donc retourné à un point antérieur et on effectuera un nouvel appel à `in-range` pour lier la variable `z`, ce qui capturera une nouvelle continuation où la valeur de y est de `2`. Finalement, le code finira par imprimer la liste `(5 3 4)` correspondant aux 3 premiers nombres respectant l'identité.

3.1.2 Modèles concrets des continuation

Les continuations sont, conceptuellement, une chaîne de *frames*. Pour illustrer le concept, nous allons montrer, dans cette section, quelques modèles très simples de représentations pour ces chaînes de *frames*. Ces chaînes seront en fait représentées par une pile de *frames*. Pour l'étude de ces modèles, nous utiliserons comme exemple une fonction d'évaluation située à la figure 3.3. Cette dernière évalue un langage acceptant comme éléments les constantes numériques, l'opération de négation à un paramètre ainsi que l'addition à deux paramètres. Dans le code de la fonction, nous avons explicitement lié les valeurs temporaires importantes à des variables pour simplifier l'explication de la structure des *frames* de continuations.

Dans cet exemple, nous n'allons considérer que les appels à eval comme des appels

de fonction. Toutefois, nous effectuerons tout de même l'addition et la négation explicitement dans nos diagrammes, mais elles seront faites directement. Les autres primitives ne seront pas considérées. Cette procédure possède donc 3 points de retour de fonctions. Dans les diagrammes, on les notera par l'abréviation P.R.

Points de retour		
Étiquette	Expression	Variables vivantes
P.R.0	`(eval '(- (+ 5 4)))`	{}
P.R.1	`(eval (cadr expr))`	{`expr`}
P.R.2	`(eval (caddr expr))`	{`n1`}
P.R.3	`(eval (cadr expr))`	{}

Pour chacune des variables vivantes à un point de retour, un espace doit être prévu dans le *frame* de continuation afin de le garder en mémoire. Un espace doit aussi être réservé pour conserver le point de retour dans le *frame* afin que la fonction appelée sache où retourner.

Dans le premier modèle que nous présentons, les *frames* ont tous la même taille et le point de retour se situe à une position prédéterminée dans un frame. Cette approche est très limitative, car, tous les *frames* ayant la même taille, on doit les allouer avec une taille correspondant à la taille maximale requise par n'importe quel appel de fonction. Dans la fonction d'évaluation, tous les *frames* doivent contenir un espace pour le point de retour. Ce point de retour sera placée dans la première cellule du *frame*. Ensuite, certains points de retour ont une variable vivante et d'autres n'en possèdent aucune. On doit donc prévoir une taille de *frame* de continuation de 2. Lors de l'appel de fonction, le *frame* de continuation sera construit et un frame d'activation contenant les paramètres d'appel sera alloué au-dessus de ce dernier. Au retour d'une fonction, la valeur au-dessus de la pile sera la valeur de retour.

Dans la figure 3.4, on voit les différents états de la pile lors de l'appel `(eval '(- (+ 5 4)))`. Pour chacun de ces diagrammes, on précise à quel moment il est situé et on décrit les *frames*.

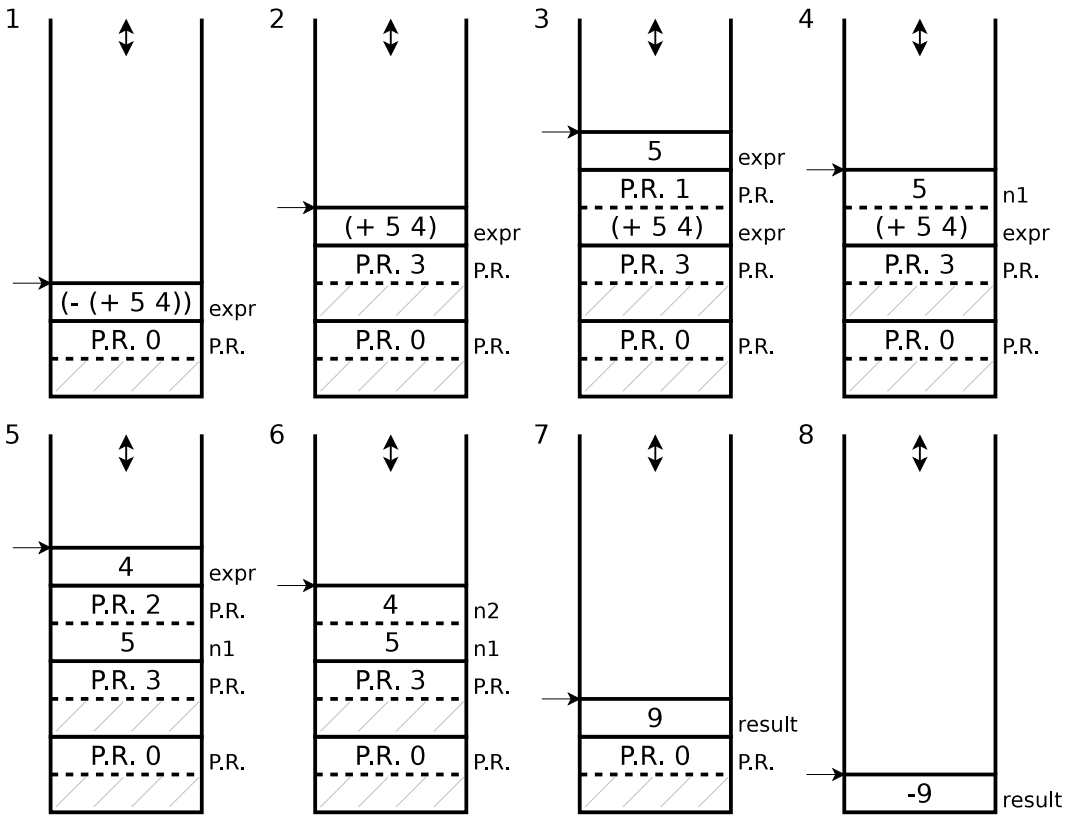

Figure 3.4 – Exemple de pile simple pour la procédure `eval`

1. Avant le premier appel à `eval`, le *frame* de continuation du P.R. 0 qui représente la fin du programme est empilé, ainsi que le paramètre `expr` contenant `(- (+ 5 4))`, l'expression à évaluer.

2. Avant l'appel à `eval` évaluant la sous expression de la négation. Le *frame* de continuation contient uniquement le point de retour 3. Le paramètre `(+ 5 4)` est par-dessus.

3. Avant l'appel à `eval` évaluant la première sous-expression de l'addition. Le *frame* de continuation contient le P.R. 1 ainsi que la variable `expr` contenant `(+ 5 4)` qui est toujours vivante au point de retour afin de pouvoir obtenir la deuxième sous-expression. Le paramètre `5` est au-dessus.

4. Après l'appel à `eval` évaluant la première sous-expression de l'addition. Le point de retour a été remplacé par la valeur de retour de la fonction `eval` : 5. L'expression `(+ 5 4)` est toujours sur la pile, car elle est nécessaire pour obtenir la

deuxième sous-expression.

5. Avant l'appel à `eval` évaluant la deuxième sous-expression de l'addition. Le *frame* de continuation contient le P.R. 2 et la valeur `5` qui est nécessaire pour pouvoir effectuer l'addition au retour. Le paramètre `4` est situé au-dessus de la pile.

6. Après l'appel à `eval` évaluant la deuxième sous-expression de l'addition. Le point de retour a été remplacé par la valeur de retour `4`. Les deux valeurs nécessaires comme paramètres à l'addition sont présentes au-dessus de la pile et l'addition est effectuée immédiatement pour ensuite retourner au point de retour indiqué dans le `frame` de continuation.

7. Après avoir effectué l'addition et retourné au P.R. 3. La valeur de retour `9` est présente au-dessus de la pile. On se trouve maintenant au point d'appel à la négation. Comme pour l'addition, la négation est effectuée sur place et on retourne directement au P.R 0.

8. Après l'appel à la négation, seule la valeur `-9` se trouve sur la pile comme valeur de retour et on est au point de retour de fin du programme.

Nous allons maintenant étendre notre modèle des *frames* de continuations en permettant le passage des paramètres et des valeurs de retour dans les registres. Pour mieux illustrer le modèle, nous n'allons permettre qu'un seul registre, R1. Les états de la pile sont à la figure 3.5.

On voit que les paramètres d'appel de fonction qui étaient, dans l'exemple précédent , placés en ordre au sommet de la pile sont maintenant positionnés différemment. Dorénavant, les premiers paramètres sont mis au sommet de la pile et le dernier paramètre dans le registre. Dans tous les appels de fonction où il n'y avait qu'un seul paramètre, il est placé dans le registre. Par exemple, lorsqu'on effectue l'addition dans le diagramme 6, l'ordre des opérandes est 5 suivi de 4 et 4 étant le dernier paramètre, il est donc dans le registre.

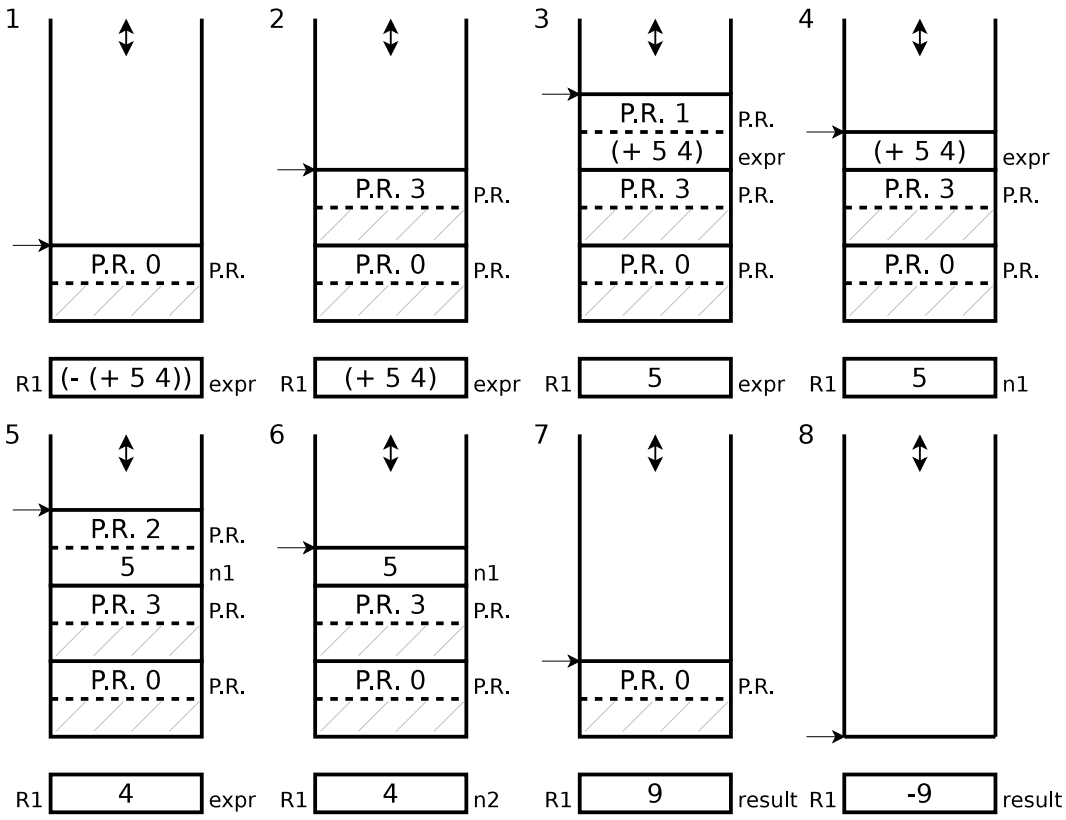

Figure 3.5 – Exemple de pile simple avec registre pour la procédure `eval`

Un autre changement avec l'exemple précédent est que les valeurs de retour, les résultats des fonctions et des primitives sont retournés directement dans le registre. On voit donc, au diagramme 7 que le résultat de l'addition, la valeur `9`, se trouve dans le registre R1.

3.2 Stratégies d'implantation

Il existe plusieurs stratégies d'implantation des continuations possédant des qualités allant de la simplicité d'implantation aux performances. Nous jugerons ces stratégies en fonction de leur vitesse et de leur efficacité.

On doit analyser la performance d'une stratégie d'implantation à plusieurs niveaux. On considère tout d'abord le coût en performance pour la création et l'élimination d'un *frame* de continuation. On devra aussi tenir compte du *surcoût*. Le surcoût est, en fait, la différence de coût pour la création d'un *frame* de continuation entre un programme

utilisant une stratégie d'implantation, mais où aucune continuation n'est capturée et un programme utilisant une implantation classique à pile et ne supportant pas les continuations de première classe. L'absence de surcoût signifie que les continuations n'occasionnent pas de pénalité dans les programmes qui ne les utilisent pas.

On doit aussi considérer d'autres coûts. Par exemple, l'usage mémoire qui sera souvent différent dans une implantation se basant sur une pile par opposition à une implantation où les *frames* doivent être explicitement chaînés. De plus, on tient compte de la pression sur le ramasse-miette. L'allocation de mémoire dans le tas causera éventuellement des ramassages et ceux-ci ont un coût en performance.

Finalement, les coûts d'usage des continuations de première classe tels que le coût de capture, le coût d'invocation et le coût de recapture sont un facteur à ne pas oublier.

Nous allons maintenant expliquer plus en détail certaines de ces stratégies d'implantation et analyser les différentes facettes de leur coût en performance.

3.2.1 Stratégie du ramasse-miette

Certaines stratégies allouent directement tous les *frames* de continuation dans le tas. Par exemple, la stratégie du ramasse-miettes consiste à allouer chaque *frame* comme un objet dans le tas et à se fier au ramasse-miette pour vérifier quels *frames* sont encore vivants et se charger de les récupérer. Le coût principal consiste au maintien du pointeur de *frame* et à l'allocation du nouveau *frame* dans le tas. Ces coûts représentent un surcoût, car ils doivent être payés même si le programme ne capture pas de continuation. De plus, d'autres coûts sont présents. Tout d'abord, le ramasse-miette doit se charger de gérer la durée de vie et de récupérer les *frames* qui ne sont plus vivants. De plus, un lien explicite doit être réservé dans chaque frame pour indiquer la suite de la continuation provoquant ainsi un coût en usage mémoire. Finalement, cette stratégie a aussi tendance à nuire aux performances de l'antémémoire, car les *frames* ne sont pas contigus en mémoire.

3.2.2 Stratégie du tas

Une autre stratégie semblable est la stratégie du tas. Elle ressemble en beaucoup de points à la stratégie du ramasse-miette mais elle garde en mémoire une liste des *frames* libres qui peuvent être réutilisés au lieu d'allouer un nouveau *frame*. En effet, lorsqu'un appel de fonction retourne, son *frame* contient un champ indiquant si celui-ci fait partie d'une continuation capturée. Si ce n'est pas le cas, il est ajouté à la liste des *frames* libres et lors d'un appel de fonction ultérieur, il peut être utilisé au lieu d'en allouer un autre. Les principaux coûts sont les mêmes que dans la stratégie du ramasse-miette et les surcoûts aussi. Les *frames* doivent aussi être explicitement liés. La différence est que la pression sur le ramasse-miette est moins élevée à cause de la réutilisation de la mémoire. Toutefois si les *frames* ne sont pas tous de la même taille, plusieurs listes doivent alors être utilisées ce qui ajoute aux coûts et augmente la complexité de l'implantation.

3.2.3 Stratégie de la pile

Une stratégie d'un type différent est la stratégie de la pile. Dans celle-ci, les *frames* de continuation sont alloués sur une pile d'exécution qui agit comme *cache* de continuation. Lorsqu'une continuation est capturée, la pile au complet est copiée dans le tas, mais reste tout de même présente dans le *cache*. Lorsqu'une continuation est invoquée, on efface le *cache* et on y recopie la continuation. Cette stratégie ne possède aucun surcoût, car, lorsqu'aucune continuation n'est capturée, les allocations de frames se feront toutes dans la même pile. Par contre, la capture prend un temps proportionnel à la taille de la continuation. De plus, la recapture prend aussi un temps proportionnel, car on recopie une fois de plus la pile.

3.2.4 Stratégie pile/tas

Enfin, il existe des stratégies hybrides. La stratégie pile/tas, comme la stratégie pile, alloue les *frames* tout d'abord dans une pile d'exécution agissant comme *cache*.

Toutefois, lorsqu'une continuation est capturée, le contenu est transféré d'un seul bloc dans le tas et la pile est immédiatement nettoyée. La pile est aussi nettoyée lorsqu'une continuation est invoquée. Comme les *frames* peuvent à la fois être consultées dans la pile et dans le tas, cette stratégie possède un surcoût, car une fonction doit toujours tester, au retour, où se situe le *frame*. La capture d'une continuation est coûteuse à cause de la copie. L'invocation et la recapture sont très rapide, car aucune copie n'est alors nécessaire.

3.2.5 Stratégie pile/tas incrémentale

Toutes ces stratégies ont un surcoût élevé ou bien un coût pour la capture, la recapture et l'invocation qui ne sont pas acceptables. C'est pourquoi le compilateur *Gambit-C* utilise une variante d'une autre stratégie. Il s'agit de l'approche pile/tas incrémentale. Dans cette approche les *frames* sont alloués sur une pile comme dans les stratégies pile et pile/tas. Toutefois, si un appel retourne dans un *frame* qui ne se trouve pas sur la pile, le système invoque une routine qui copie le prochain *frame* du tas vers la pile. Donc, tous les appels se font dans la pile. Cette stratégie n'a donc aucun surcoût. La capture doit transférer les *frames* vers la pile comme la stratégie pile/tas et la performance est comparable. Le seul point de perte de performance est la copie d'un *frame* du tas vers la pile. On verra plus en détail comment elle est implantée dans les prochaines sections.

3.3 Appels de fonctions

Dans *Gambit-C*, les *frames* de continuation sont alloués de manière contiguë dans une pile d'exécution. Les points de retour ainsi que les informations pertinentes sur ces *frames* sont contenues dans des structures appelées *descripteurs de points de retour* (P.R.). Ces structures sont des objets *Scheme* alloués de manière statique lors de l'initialisation d'un module. Ils contiennent, entre autres, un champ appelé *link* qui indique où se situe, dans le frame, le point de retour dans l'appelant ainsi que le champ *fs* indiquant la taille du frame. Finalement, ils comportent un pointeur vers l'endroit dans

```
(define (g x y)
  (+ x y))

(define (f a b)
  (+ (g 1 a) b))

(f 2 3)
```

Figure 3.6 – Exemple simple d'appel de fonction

le code où se situe le point de retour (non représenté dans les diagrammes).

Lors de l'appel de fonction, la fonction appelante va allouer un frame de continuation au sommet de la pile d'exécution. Le compilateur aura prévu un emplacement pour le P.R. de l'appelant, sans considération spéciale, et aura indiqué cet endroit dans le champ *link* du P.R. de l'appelé. Le P.R. de l'appelant sera obtenu dans le registre R0 et placé à cet endroit. Les variables, possiblement temporaires créées par le compilateur, dont la fonction appelante pourrait avoir besoin au retour seront placées dans ce dernier. Un certain nombre de paramètres d'appel seront placés dans les registres R1 et suivants. Ce nombre dépend de la plateforme pour laquelle le code est compilé. Le reste des paramètres seront placés au-dessus de la pile (au-dessus du frame de continuation). Les valeurs dont la fonction appelante pourrait avoir besoin au retour seront aussi placées dans ce dernier. Finalement, juste avant l'appel, on placera le descripteur du point de retour dans le registre R0 et on effectuera le saut vers le code de la fonction appelée.

Pour expliquer ces concepts, nous utiliserons un exemple simple d'appel de fonction situé à la figure 3.6. Dans la figure 3.7, on voit l'état des structures juste après le saut au code de la fonction `f`. Les deux paramètres, `2` et `3`, sont présents dans les registres `R1` et `R2` et le P.R. 0 qui est le point de retour marquant la fin du programme, est dans le registre `R0`. Ce P.R. possède un *fs* de 0 car aucune variable n'est vivante à ce point et il n'y a pas d'appelant.

Dans la figure 3.8, on se trouve immédiatement après le saut dans le code de la fonction `g`. Le frame de continuation de l'appel a `g` se trouve au sommet de la pile et

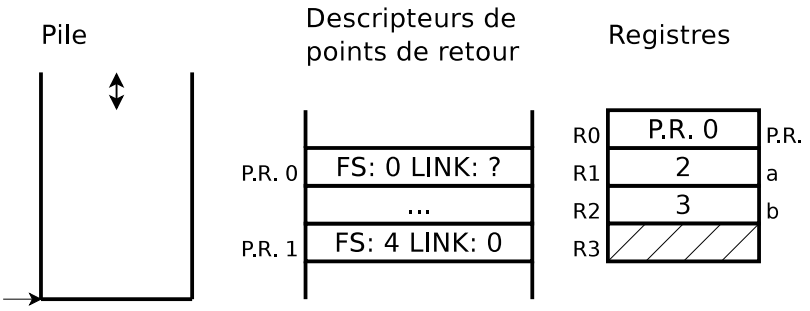

Figure 3.7 – État des structures avant l'appel de fonction

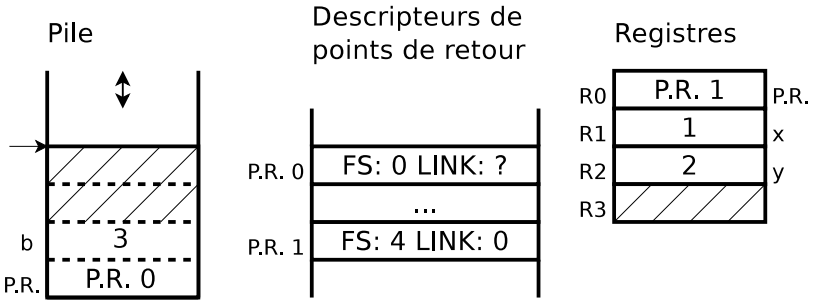

Figure 3.8 – État des structures après l'appel de fonction

contient dans son emplacement 0 le P.R. de l'appelant de `f`, c'est à dire P.R. 0 ainsi que la valeur de `b` dans son emplacement 1 car b est nécessaire pour effectuer l'addition au retour de `g`. Le P.R. 1 se trouve dans le registre `R0` et représente le P.R. de l'appel à `g`. Finalement les deux paramètres de l'appel à `g` sont dans les registres `R1` et `R2`.

3.4 Capture de continuations

Dans *Gambit-C*, la capture de continuation se fait à l'aide de la procédure `continuation-capture`. Elle prend en paramètres une méthode à laquelle elle passera la continuation capturée en paramètre ainsi qu'un nombre arbitraire d'autres paramètres qu'elle repassera tels quels. La continuation capturée est encapsulée dans un objet de type continuation. Pour invoquer cette dernière, on doit appeler la procédure `continuation-return` avec la continuation comme paramètre ainsi que la valeur de retour à passer à celle-ci. `Call/cc` peut être implanté à l'aide de continuation-capture comme à la figure 3.9

```
(define (call/cc f)
  (continuation-capture
    (lambda (k)
      (f (lambda (r)
           (continuation-return k r))))))
```

Figure 3.9 – Implantation de call/cc

3.4.1 Break Handler et Break Frame

Dans *Gambit-C*, la capture de continuation alloue directement la continuation sur la pile. Pour cette raison, la pile se trouve à être *brisée* lorsqu'une continuation est capturée. Lorsque cela arrive, le *frame* d'un appelant ne se trouvera pas directement sous le *frame* de son appelé. C'est à ce moment que se déroule le travail du *break handler*.

Lorsque le *frame* de l'appelant ne suit pas directement, un *break frame* suit sur la pile et le P.R. de ce frame est l'entrée dans le *break handler*. Le *break frame* agit donc comme frame de continuation de ce dernier. Le *break frame* contient un lien vers le frame de l'appelant et c'est le travail du *break handler* de recopier ce frame vers le sommet de la pile et de continuer l'exécution.

3.4.2 Routine de capture

La routine de capture de continuation, dont le code est présenté à la figure 3.10, va placer le P.R. courant dans une cellule prévue à cet effet dans le *frame* courant et placer le P.R. du break handler dans le registre R0. Elle alloue ensuite une continuation directement sur la pile. Une continuation nécessite trois mots mémoire, mais quatre sont alloués pour satisfaire les contraintes d'alignement. Elle y place l'en-tête de l'objet, suivi d'une référence au *frame* courant et enfin l'environnement dynamique du *thread* courant. Finalement, elle alloue le *break frame* et ajuste sa référence pour pointer vers le *frame* courant.

```
SCMOBJ frame;
SCMOBJ cont;
SET_STK(-FRAME_STACK_RA,R0)
R0 = GSTATE->handler_break;
frame = CAST(SCMOBJ,fp);

// Allocation de l'objet continuation sur la pile
ADJFP(ROUND_TO_MULT(SUBTYPED_OVERHEAD+CONTINUATION_SIZE,FRAME_ALIGN))

// Creation de l'en-tête de la continuation et assignation aux champs
SET_STK(0,MAKE_HD_WORDS(CONTINUATION_SIZE,sCONTINUATION))
SET_STK(-1,frame)
SET_STK(-2,FIELD(ps->current_thread,THREAD_DENV))
cont = TAG(&STK(0),tSUBTYPED);

// Allocation du break frame sur la pile
ADJFP(BREAK_FRAME_SPACE)
SET_STK(-BREAK_FRAME_NEXT,frame)

ps->stack_break = fp;
```

Figure 3.10 – Code de `(continuation-capture f)`

3.4.3 Exemple de capture

Pour illustrer la capture de continuation, nous analysons un exemple semblable à celui utilisé dans la section sur l'appel de fonction. L'exemple est à la figure 3.11. La fonction `f` appelle une fois de plus la fonction `g` mais cette fois `g` capture la continuation à son point d'entrée. La procédure `continuation-capture` se charge de capturer la continuation et d'appeler une méthode avec cette dernière comme paramètre. Dans le code, on voit une deuxième version de la fonction `g`. Dans cette version, la `lambda` a été soumise au *lambda lifting*, c'est-à-dire que sa signature est modifiée pour que ses variables libres lui soient passées en paramètre et ainsi on évite de créer une fermeture.

Le premier diagramme à la figure 3.12 représente l'état de la pile et des structures juste après l'appel à `g` dans la fonction `f`. Les paramètres de `g` sont dans les registres `R1` et `R2`. Le P.R. 1, qui correspond au retour de `g`, est dans le registre `R0`. La valeur de `b` qui est nécessaire pour effectuer l'addition au retour de `g` est dans le *frame* de

```
(define (g x y)
  (continuation-capture
   (lambda (k)
     (+ x y))))

;; (define (g x y)
;;   (continuation-capture
;;    (lambda (k x y)
;;      (+ x y))
;;    x
;;    y))

(define (f a b)
  (+ (g 1 a) b))

(f 2 3)
```

Figure 3.11 – Exemple de capture de continuation

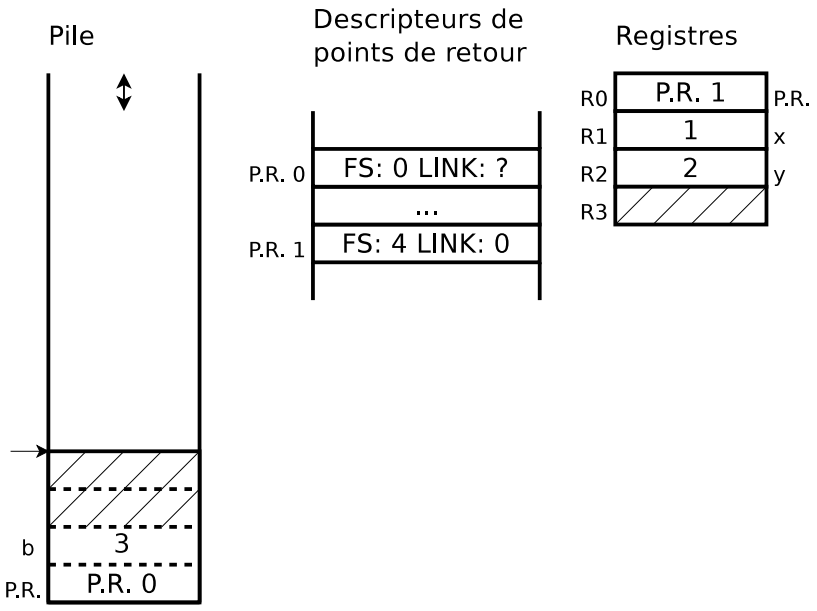

Figure 3.12 – État des structures avant la capture de la continuation

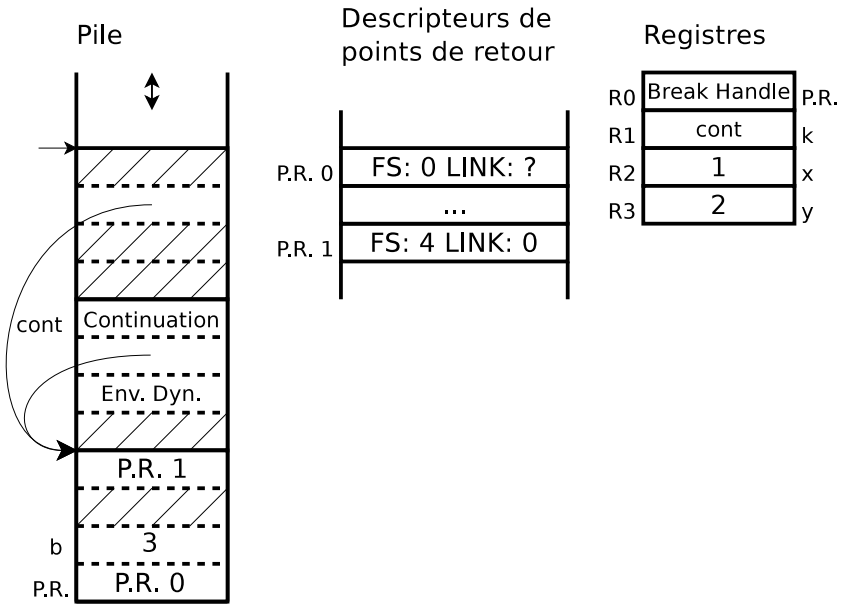

Figure 3.13 – État des structures après la capture de la continuation

continuation.

Le deuxième diagramme à la figure 3.13 représente l'état après la capture de la continuation, juste avant l'appel à la `lambda`. Les trois paramètres qui devront être passés à cette dernière sont dans les registres `R1` à `R3`. Il s'agit de la continuation et des variables `x` et `y`. De plus, la continuation a été allouée sur la pile et pointe vers le *frame* de continuation correspondant au retour de la fonction `g`. Au-dessus de la continuation se trouve le *break frame* ou *frame* de continuation du *break handler*. Il contient tout simplement un lien vers le même *frame* que la continuation. Le P.R. qui se trouve dans le registre `R0` est le point de retour permettant l'entrée dans le *break handler*. Finalement, le P.R. 1 est placé dans la première cellule du frame de continuation du retour de `g` parce-que ce frame est le premier de sa section de pile.

3.5 Exécution du Break Handler

Lorsqu'un appel retourne à un parent dont le *frame* de continuation n'est pas au sommet de la pile, le *break handler* est invoqué. Ce *frame* peut se trouver plus bas dans la pile ou bien être présent sous forme d'un objet dans la pile. Pour chaque cas, une routine différente est exécutée par le *break handler*.

```
(define (h)
  (continuation-capture
    (lambda (k)
      1)))

(define (g x y)
  (+ (h) x y))

(define (f a b)
  (+ (g 1 a) b))

(f 2 3)
```

Figure 3.14 – Exemple de retour par break handler

Nous allons montrer deux exemples de retour à un *frame* qui n'est pas au sommet de la pile. Dans un cas, il se trouvera plus loin sur la pile, dans l'autre, il sera contenu dans un objet sur le tas. Nous allons utiliser un exemple semblable à celui utilisé pour illustrer la capture de continuation. Il est situé à la figure 3.14. Nous avons ajouté un niveau de plus d'appels de fonction pour mieux illustrer le concept et n'avons pas mis de paramètres à la fonction `h` afin d'éviter le *lambda lifting* et ainsi simplifier l'exemple.

3.5.1 Code du break handler

La routine du break handler, dans le cas où le *frame* est situé dans la pile, est montré à la figure 3.15. Elle commence par obtenir le P.R. du *frame* à copier dans la première cellule de ce dernier. Elle consulte ensuite ce descripteur pour obtenir la taille du *frame* et l'endroit dans le *frame* ou se trouve le P.R. de son appelant, le *link*. Elle entame ensuite la copie de chacune des cellules du *frame* vers le sommet de la pile puis consulte la cellule *link* pour obtenir le P.R. de l'appelant et le placer dans la cellule réservée du *frame* de l'appelant. Elle ajuste ensuite le *break frame* pour qu'il pointe maintenant vers le *frame* de l'appelant. Finalement, elle change la cellule *link* du *frame* fraichement copié pour qu'elle contienne le P.R. du *break handler*.

Lorsque le frame est dans le tas, il est contenu dans un objet Scheme et contient son

```
// Nous obtenons le frame de l'appelant dans le break frame
cf = ___STK(-___BREAK_FRAME_NEXT);
fp = CAST(SCMOBJ*,cf);
// Nous obtenons le P.R. du frame
ra1 = FP_STK(fp,-FRAME_STACK_RA);
// Taille du frame et position du link
RETN_GET_FS_LINK(ra1,fs,link)
// Positionnement au bas du frame
FP_ADJFP(fp,-FRAME_SPACE(fs));
// Copie du frame au sommet de la pile
for (i=fs; i>0; i--)
  SET_STK(i,FP_STK(fp,i))
// Reajustement du prochain frame appelant et du break frame
ra2 = STK(link+1);
FP_SET_STK(fp,-FRAME_STACK_RA,ra2)
SET_STK(-BREAK_FRAME_NEXT,CAST(SCMOBJ,fp))
SET_STK(link+1,GSTATE->handler_break)
```

Figure 3.15 – Code du break handler dans le cas où le frame est dans la pile

```
// On obtient le corp de l'objet frame dans le tas
cf = ___STK(-___BREAK_FRAME_NEXT)
fp = BODY_AS(cf,tSUBTYPED);
// On recupere le P.R. du frame ainsi que la taille et le link
ra1 = fp[FRAME_RA];
RETN_GET_FS_LINK(ra1,fs,link)
// On copie au sommet de la pile à partir de la base
fp += fs+1;
for (i=fs; i>0; i--)
  SET_STK(i,FP_STK(fp,i))
// Réajustement du break frame et du frame copié
SET_STK(-BREAK_FRAME_NEXT,STK(link+1))
SET_STK(link+1,GSTATE->handler_break)
```

Figure 3.16 – Code du break handler dans le cas où le frame est dans le tas

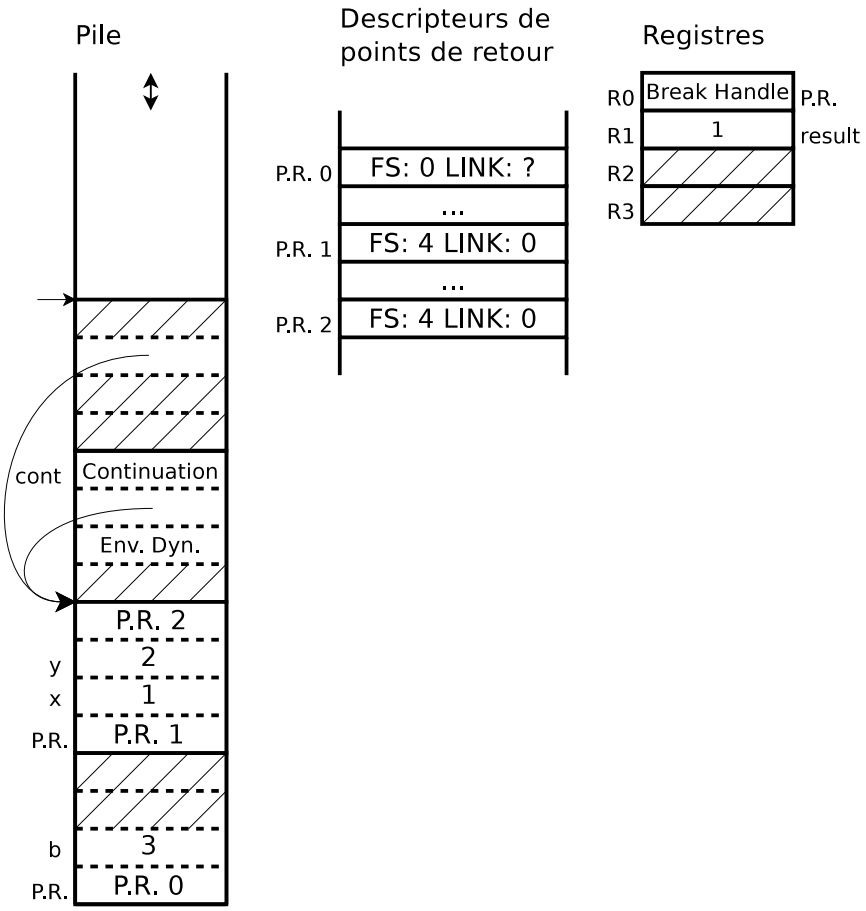

Figure 3.17 – État des structures avant le *break handler*

propre P.R. contrairement aux *frames* de la pile. Les *frames* du tas contiennent aussi une référence vers le prochain *frame* dans leur cellule *link* au lieu du P.R. du *frame* de l'appelant. On voit ces différences dans le code situé à la figure 3.16. La principale différence entre le code pour un *frame* sur la pile et le code pour un *frame* sur le tas est qu'on a pas besoin d'ajuster le nouveau *frame* suivant le *break frame* avec son P.R. car les *frames* sur le tas contiennent tous explicitement leur point de retour. De plus, lorsqu'on ajuste le *break frame* avec le prochain *frame* on obtient la référence dans la cellule *link* du *frame* courant au lieu de simplement consulter le pointeur de pile.

3.5.2 Exemple d'exécution du break handler

Dans la figure 3.17, on se trouve juste avant le retour de la procédure `h` à la procédure `g`. Sur la pile, on voit les *frames* de continuation de `f` et de `g` ainsi que la continuation

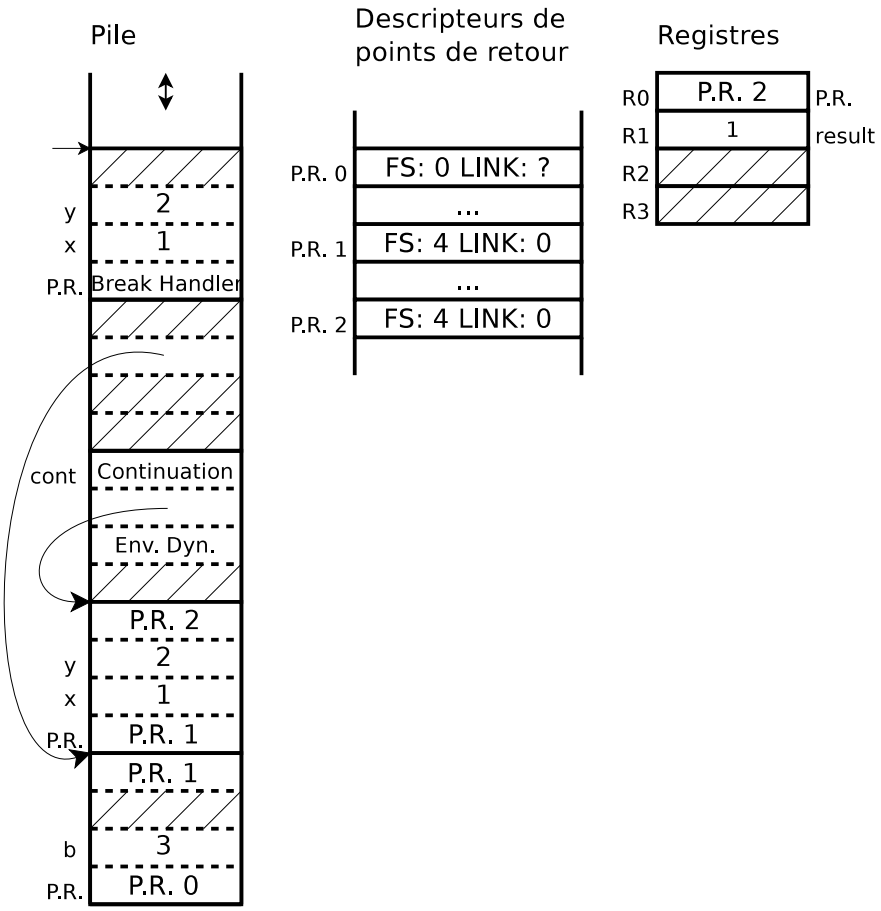

Figure 3.18 – État des structures après le *break handler*

capturée et le *break frame*. Dans le registre `R0` se trouve le P.R. du *break handler* et dans le registre `R1` la valeur de retour de la méthode `h`.

Après l'exécution du *break handler*, les structures sont dans l'état illustré à la figure 3.18 et on se trouve au retour de la fonction `h`. Le frame de continuation de la procédure `g` a été copié au sommet de la pile et on a ajusté sa cellule *link* pour contenir le P.R. du *break handler*. Le *break frame* a été mis a jour pour faire référence au *frame* de continuation de la méthode `f`. Enfin le P.R. 2 est dans le registre `R0`.

La figure 3.19 représente les structures si le *frame* est sur le tas. La principale différence que l'on remarque est que la continuation et les deux *frames* de continuation sont dans des objets sur le tas. On voit aussi que le *frame* de continuation de `g` est explicitement lié au *frame* de continuation de `f`.

Après l'exécution du *break handler*, on a tout simplement ajusté la référence du

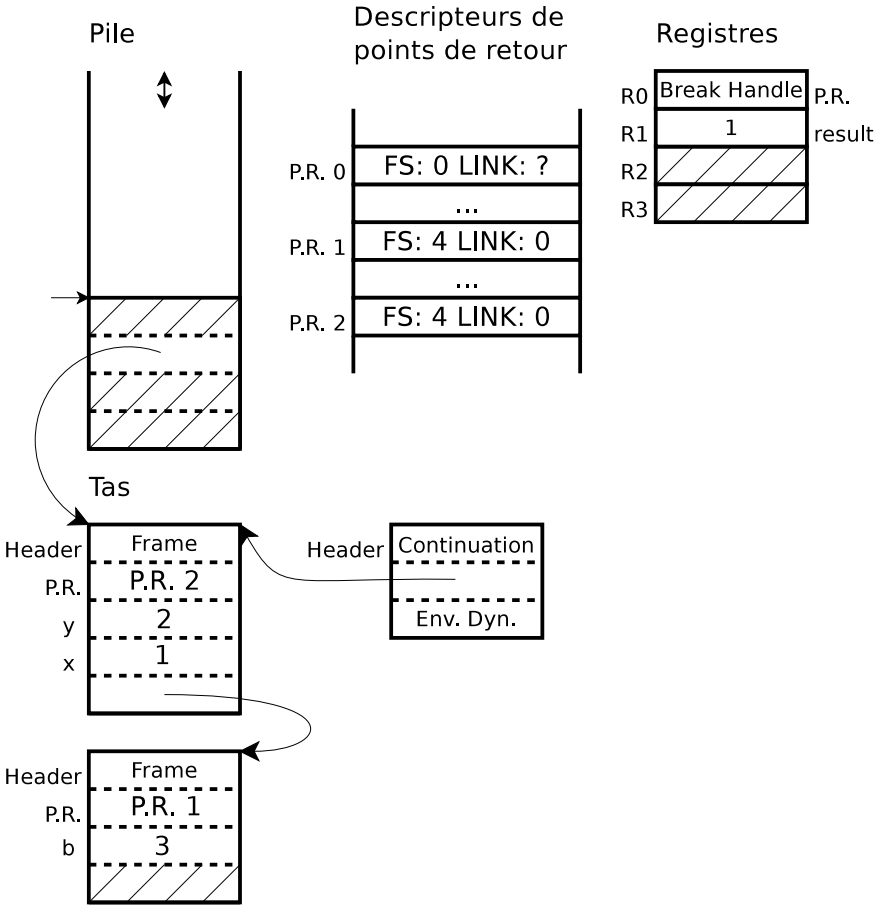

Figure 3.19 – État des structures avant le break handler

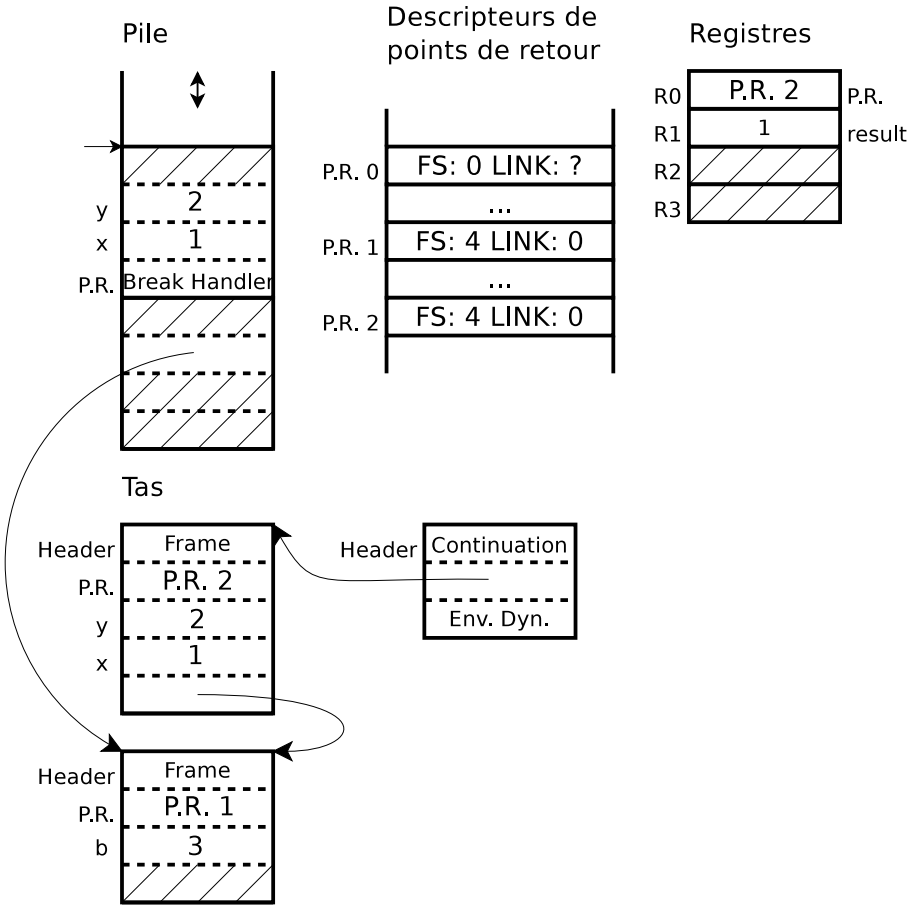

Figure 3.20 – État des structures après le break handler

break frame vers le *frame* suivant. Comme dans le cas de la pile, le *frame* a été copié sur la pile en ne copiant pas l'en-tête de l'objet. L'état des structures est illustré à la figure 3.20

CHAPITRE 4

NOTRE SYSTÈME

Ce chapitre explore l'implantation d'un dialecte à modèle partitionné de *Multilisp* étendu avec des opérations de communication par passage de messages et utilisant la technique de la *création paresseuse de tâches* pour le balancement de charge dynamique.

4.1 Création Paresseuse de Tâches

La technique de la *Création Paresseuse de Tâches* est une méthode de partitionnement dynamique et d'ordonnancement de tâches efficace pour les applications à fine granularité. Elle a été introduite dans [20] où était présentée une implantation bâtie sur des systèmes à mémoire partagée efficace. Une variante à passage de message, provenant de [13], est l'inspiration pour l'implantation que nous développons dans ce mémoire.

4.1.1 Ordonnancement

L'une des responsabilités du système consiste à ordonnancer les tâches à exécuter, c'est-à-dire choisir dans chaque processus, lorsque c'est nécessaire quelle tâche sera exécutée. Un système préfèrera toujours l'exécution d'une tâche locale pour aider la localité et réduire le transfert de données ainsi que la pression sur les autres processeurs. Lorsqu'aucun travail n'est disponible localement, le processeur tentera d'effectuer un *vol de tâche*, c'est-à-dire contacter un autre processeur pour obtenir une tâche pouvant être exécutée. Le vol de tâche est le seul moyen de distribuer automatiquement le travail dans notre système. Localement, il y a deux moments où un processeur devra prendre une décision concernant la prochaine tâche à exécuter.

Tout d'abord, lors de la création d'une tâche par l'insertion d'un `future`, il est possible d'exécuter la tâche enfant ou la tâche parente. La politique de la création paresseuse de tâches est d'exécuter la tâche enfant en premier et de suspendre la tâche parent. L'avantage de cette décision est de réduire la probabilité de suspension de la

tâche parente sur la promesse de la tâche enfant si celle-ci n'a pas terminé lorsque le résultat sera nécessaire.

Une autre décision doit être prise lorsqu'une tâche termine. La politique de base est d'exécuter immédiatement le parent de la tâche si cela est possible. La combinaison de ces deux politiques permet d'assimiler l'exécution des tâches à un comportement de pile ce qui est essentiel à l'implantation efficace de la création paresseuse de tâches.

Cela permet d'utiliser directement la pile d'exécution pour représenter la pile de tâches. Il nous est possible de simplement représenter les frontières des tâches par une annotation sur les `frames` de continuation. Ainsi, l'ordre d'exécution du programme sur un seul processeur sera le même que si les `futures` n'étaient pas présents.

4.1.2 Représentations et vol des tâches

Dans une implantation naïve de Multilisp, le système n'a aucune décision à prendre quant au partitionnement, car il est entièrement déterminé par le positionnement des `futures` dans l'application. Toutefois, comme indiqué précédemment, les décisions de partitionnement peuvent de pas concerner uniquement la création de tâches, mais aussi le choix de représentation.

C'est sur ce principe que se base la création paresseuse de tâches. Le système ne prend aucune décision sur la création des tâches, mais uniquement sur le choix de représentation pour ces dernières. Deux types de représentations sont utilisés, la représentation légère qui est peut coûteuse et permet une fine granularité dans les programmes. La seconde est la représentation lourde qui est utilisée lors du vol de tâches.

Dans l'implantation originelle de Feeley [13], les deux représentations contiennent comme élément de base une continuation représentant l'état de la tâche. L'implantation de ces dernières est expliquée en détail dans le chapitre 3. Les tâches légères consistent simplement en une file de pointeurs indiquant les frontières des tâches dans la pile d'exécution.

Notre implantation, que Feeley avait nommée la *Création très Paresseuse de Tâches*

dans sa dissertation [13], simplifie cette approche en laissant tomber la file de pointeurs et en indiquant tout simplement les frontières des tâches grâce à un marquage des *frames* de continuation. Cela réduit la quantité de travail requise pour créer une tâche légère, mais elle est plus coûteuse lors du vol de tâche, car la pile d'exécution doit être parcourue afin de trouver les tâches.

Les tâches légères sont converties en tâches lourdes lors du vol de tâche. On choisit toujours la tâche la plus ancienne, c'est-à-dire la plus profonde dans la pile d'exécution. C'est ce qu'on appelle le *vol par le bas*. Une tâche lourde est créée en capturant la continuation de la tâche légère qu'on souhaite voler. Pour chaque création de tâche lourde, une promesse doit aussi être créée et associée à la tâche. Le tout est encapsulé dans une fermeture qui peut être passée au processus voleur qui pourra alors lancer cette tâche en exécutant la fermeture. Les détails de cette implantation sont décrits plus loin.

4.2 Fonctionnement du système

Notre système est divisé en deux parties importantes distinctes. Il s'agit tout d'abord des programmes parallèles qui seront exécutés sur chacun des processeurs et ensuite, du gestionnaire global, un processus système séparé, qui se charge de démarrer, surveiller et gérer les processus exécutant ces programmes sur chacun des processeurs utilisés.

Les programmes sont codés à l'aide d'une librairie contenant diverses primitives de communication et de contrôle rappelant ce qui est fourni par *MPI*. Ces opérations couvrent plusieurs structures différentes de communication. Par exemple, l'opération *broadcast* permet de transmettre une donnée d'un processus vers tous les autres.

Les programmes sont aussi démarrés à l'intérieur d'un environnement contenant toute la machinerie nécessaire à l'exécution de parties de code contenant des *futures*. Dans ces régions du programme qui doivent être explicitement identifiées, le transfert des données et des tâches se fait automatiquement grâce au système de création paresseuse de tâches et à la sérialisation des continuations.

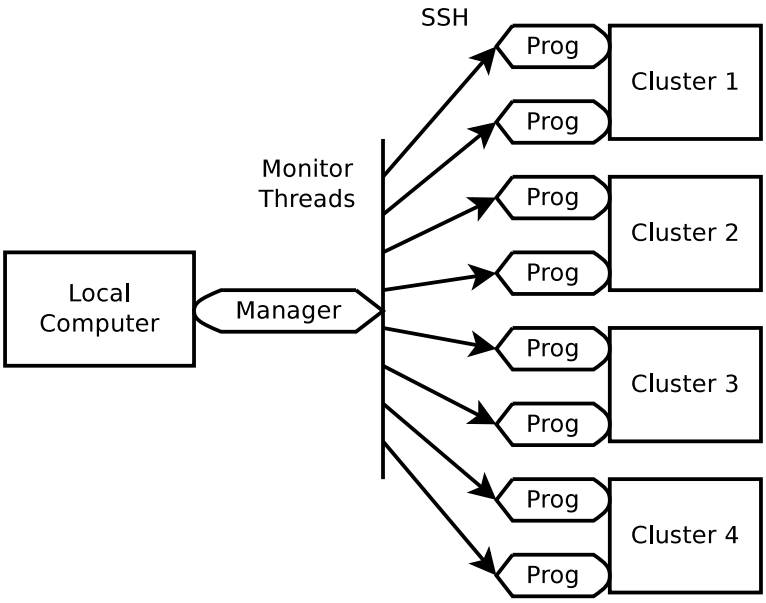

Figure 4.1 – Représentation du système pour un cluster de 4 noeuds à 2 cpus.

Quant au gestionnaire global, il démarre un processus Gambit exécutant le programme désiré pour chaque processeur qu'on veut utiliser. Il surveille ensuite ces processus en temps réel afin de pouvoir faire terminer l'ensemble si une condition d'erreur se produit. Une autre de ses fonctions est de transmettre les paramètres d'exécution aux processus et surtout de synchroniser les processus pour éviter les conditions de course dans leur démarrage et leur fin.

On peut voir une représentation du système à la figure 4.1. Dans ce diagramme, le gestionnaire a démarré le système dans une grappe de quatre noeuds possédant chacun 2 processus. Pour chacun des processus devant être exécutés, 8 en tout, un thread a été créé dans le gestionnaire afin de surveiller l'état. Ensuite, une connexion ssh par processus a été ouverte afin de démarrer le processus sur le noeud distant.

4.3 Programmes

Un des aspects importants de la création de programmes parallèle est la facilité d'introduction du parallélisme. Lors de la création de tels programmes dans un contexte distribué, quelques facteurs doivent être considérés. La première différence importante avec le parallélisme dans un contexte de mémoire partagée est que les processeurs

possèdent des mémoires distinctes et sont séparés les uns des autres. Il doit donc exister une manière pour ces processeurs de se communiquer de l'information.

La *librairie de communication* est la partie de notre système qui occupe cette fonction. Elle est bâtie sur le principe du passage de messages un peu à la manière du *Message Passing Interface*. Elle fournit une interface qui permet la communication d'un pair à un autre ainsi que les communications collectives dites *broadcast*.

Le concept de communication collective est très utile dans le cadre distribué, car les structures de données nécessaires au calcul ne sont pas présentes de manière uniforme sur les processeurs. Il permet alors de n'avoir qu'à écrire un seul programme alors que différents processeurs devront accomplir des tâches variées. Cela est essentiel pour permettre la création aisée de programmes parallèles.

Un autre aspect fondamental du calcul parallèle qui n'est habituellement pas pris en charge par les systèmes distribués est le balancement de charge. Dans certains cas, par exemple la multiplication de matrices, la manière de diviser le travail entre les processeurs est évidente. En effet, on peut diviser manuellement la tâche en plusieurs sous-tâches de taille égale et qui prendront toutes le même temps à s'accomplir.

Dans d'autres cas, la division est beaucoup moins simple. Le travail se divise en sous-tâche mais elles ne comportent pas toutes la même quantité de calculs. Cela a pour effet que certains processeurs auront terminé bien avant d'autres et que l'on perdra de la capacité de calcul. Il est alors nécessaire d'introduire une manière de pouvoir distribuer du travail aux processeurs automatiquement en fonction des besoins.

Dans les programmes distribués, cela est souvent accompli en donnant pour tâche à un processeur de fournir les autres en travail. Cette approche n'est pas optimale, car on perd alors la contribution au travail d'un des processeurs et on ajoute à la tâche du programmeur, car il devra coder la logique du distributeur différemment des autres processeurs.

Nous approchons cette problématique en adaptant une technique provenant du parallélisme sur les processeurs à mémoire partagée, le concept de *futures* de *Multilisp*.

```
(define (main)
  (if (= 0 node-id)
      (! (node 1) "hello world!")
      (print (?))))
```

Figure 4.2 – Exemple de programme simple avec deux processeurs

Le système permet de déclarer certaines zones du programme où le partage de tâches sera géré par l'implantation de la création paresseuse de tâches. Sur chaque processeur, un gestionnaire de tâches est démarré et a pour fonction de démarrer et surveiller un processus travailleur et de voler du travail à ses pairs afin de garder le processeur chargé en tout temps.

Dans cette section, nous allons tout d'abord présenter plus en détail la librairie de communication et ensuite nous aborderons comment nous implantons Multilisp et la création paresseuse de tâches et les problèmes qui sont rencontrés lors de son utilisation dans un contexte distribué.

4.3.1 Librairie de communication

La librairie de communication est la partie du système qui permet aux processeurs de s'échanger de l'information arbitraire d'un pair à un autre ou de manière collective. Elle est implantée à l'aide du système acteur Termite [14] ce qui permet de réutiliser toute la machinerie de communication que ce système met en place. La communication ressemble donc sur plusieurs points à la manière dont elle est faite en Termite.

Tout d'abord, la communication entre les *threads* se fait à l'aide de la primitive "`!`" qui permet d'envoyer un message à un autre *thread*. Notre système ajoute une autre primitive `(node n)` qui permet d'obtenir une référence vers le *thread* fondamental situé sur le $n^{\text{ème}}$ processeur. Par exemple, à la figure 4.2, on voit un exemple de programme parallèle à deux processeurs où le premier processeur envoie un message au deuxième processeur qui l'imprime ensuite à sa sortie standard.

La librairie supporte aussi la communication collective à l'aide de la fonction `broad-`

```
(define (broadcast object root)
  (if (= node-id root)
      (begin
        (for-each (lambda (peer) (! peer object)) peer-mains)
        object)
      (?)))
```

Figure 4.3 – Exemple d'implantation de l'opération broadcast

`cast` dont une implantation simple est montrée à la figure 4.3. On appelle la procédure `broadcast` avec un objet arbitraire à envoyer ainsi que l'identité du processeur qui enverra à tous les autres, `root`. La procédure `?` est simplement la fonction de réception. Dans ce code, la variable `peer-mains` contient une liste des *threads* roulant la fonction `main` sur chacun des processeurs, donc le fil principal d'exécution.

4.3.2 Partage de tâches

Dans les programmes distribués, le partitionnement de tâches se fait habituellement de manière statique. On trouve une manière de diviser le problème en sous-parties ayant sensiblement la même charge de travail et on distribue ces sous-tâches aux processeurs qui forment notre système.

Pour certains problèmes, c'est facile. Par exemple, dans le cas de la multiplication de matrices, on peut diviser la première matrice en sous-ensembles contenant tous le même nombre de lignes. Chacun de ces sous-ensembles devrait prendre environ le même temps d'exécution et on devrait obtenir un facteur d'accélération presque linéaire.

Pour d'autres problèmes, c'est plus difficile. Dans le cas de mandelbrot, la quantité de travail à faire pour un point donné est dépendante de sa position. Si on divise le travail en sous-ensembles de lignes, certains processeurs termineront bien avant d'autres. Il est toutefois possible d'être plus astucieux et de séparer le travail en ensembles de lignes contiguës, mais plutôt en ensembles contenant des lignes séparées par un intervalle fixe. On fait ce choix en réalisant que deux lignes situées une à la suite de l'autre ont généralement une quantité de travail assez semblable. En distribuant les lignes de cette

façon, on obtient des ensembles comportant sensiblement la même quantité de travail.

Finalement, certaines tâches ne comportent pas de division statique claire. Un exemple bien connu est le problème des *n-queens* où l'on tente de déterminer combien de solutions il existe au *n-queens puzzle*. Une solution est un échiquier de NxN comportant N reines qui ne se trouvent pas sur la même ligne, colonne ou diagonale. Ce problème ne peut être séparé statiquement, car il s'agit essentiellement d'une recherche dans un arbre qui n'est pas balancé. (Algorithme par retour arrière)

4.4 Implantation de la Création très Paresseuse de Tâches

Notre implantation de la création très paresseuse de tâches repose en grande partie sur la conception des continuations dans le compilateur *Gambit-C*. Ce sujet est couvert en détail dans le chapitre 3. Il est aussi dépendant de la sérialisation des continuations.

Dans cette section, nous allons tout d'abord présenter le système de *gestionnaire des travailleurs* qui offre la machinerie de communication et de gestion permettant le vol, la transmission et la surveillance des tâches. Ensuite, nous présenterons comment les continuations sont utilisées afin d'accomplir le vol de tâches.

4.4.1 Gestionnaire des Travailleurs

Le *gestionnaire des travailleurs* a comme principale tâche de démarrer et surveiller un *thread* exécutant une tâche. Il se charge aussi d'obtenir des tâches afin de redémarrer son travailleur lorsque celui-ci termine. Finalement, il doit répondre aux demandes des autres gestionnaires lorsque ceux-ci désirent obtenir du travail, c'est-à-dire voler une tâche. On voit une illustration de cette organisation dans la figure 4.4.

Il est implanté à l'aide d'un processus léger Termite pouvant se trouver dans deux états possibles, en attente ou non, et qui doit répondre à un certain nombre de messages possibles. Aucun des messages envoyés par un gestionnaire ne provoque une attente active. Le gestionnaire doit toujours être en mesure de répondre aux messages des autres sinon des cas d'étreintes fatales pourraient survenir. C'est pourquoi les réponses

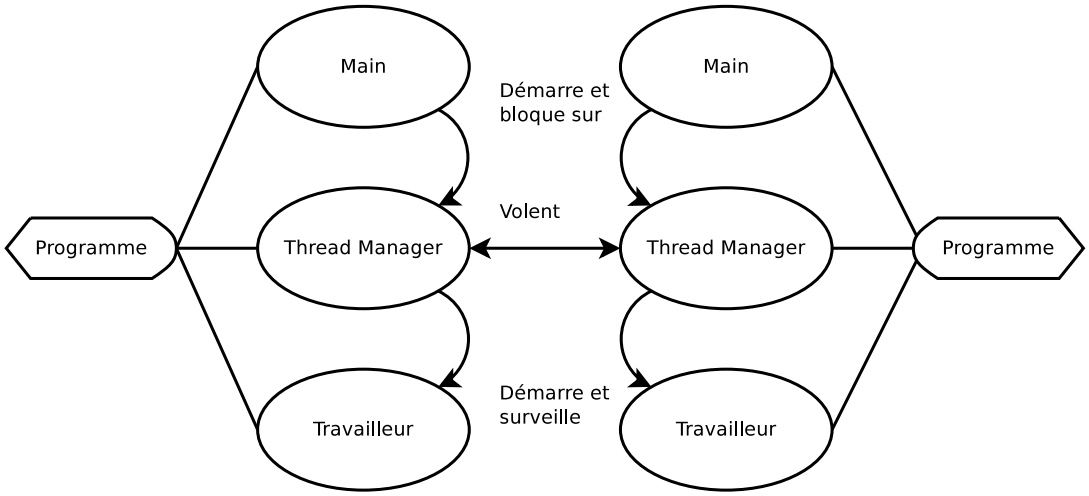

Figure 4.4 – Organisation du thread-manager

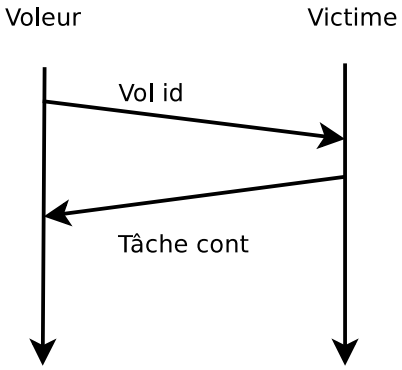

Figure 4.5 – Illustration d'un vol ayant réussi

à certains messages ont des numéros de série afin de les identifier.

1. **Message de vol** : Le message de vol parvient au gestionnaire victime avec un seul paramètre, l'identité du gestionnaire voleur. Il vérifie alors s'il possède une tâche lourde déjà prête à être exécutée et sinon, il tente de voler une tâche dans la pile d'exécution de son travailleur. Il renvoie ensuite un *Message de tâche* au demandeur contenant le travail à effectuer ou bien une indication de l'échec du vol le cas échéant. On peut voir une illustration de ces communications dans les figures 4.5 et 4.6.

2. **Message de tâche** : Le message de tâche est renvoyé en réponse au message de vol et contient la tâche à effectuer qui sera alors immédiatement exécutée dans

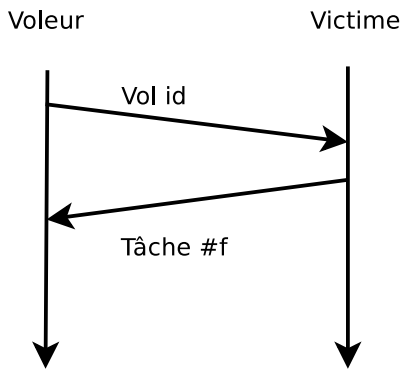

Figure 4.6 – Illustration d'un vol ayant échoué

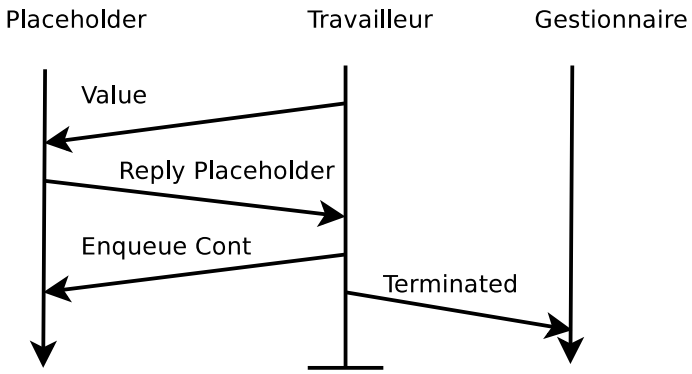

Figure 4.7 – Illustration de la fin d'un travailleur sur un placeholder non déterminé

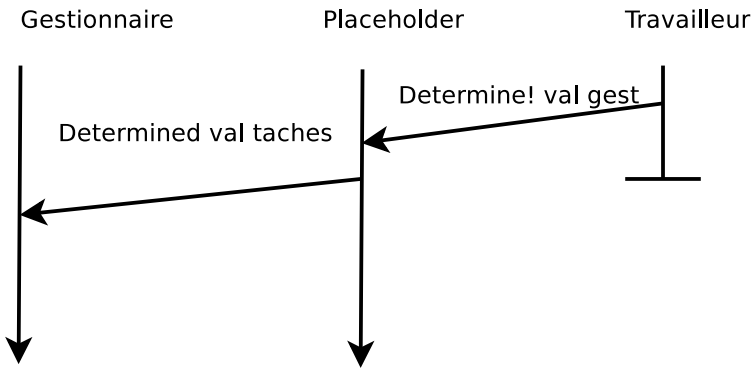

Figure 4.8 – Illustration de la fin d'un travailleur lorsqu'il détermine un placeholder

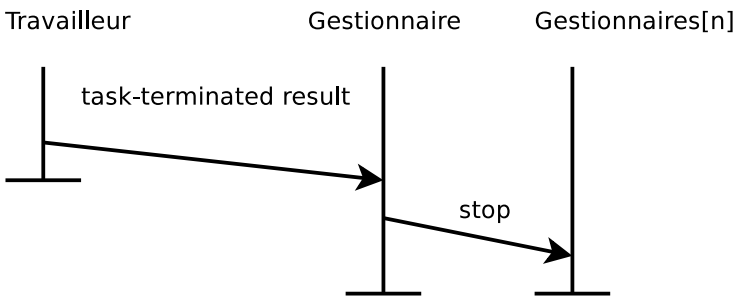

Figure 4.9 – Illustration de la fin de la tâche globale

un travailleur avant que le gestionnaire ne se remette en attente de messages. Si le vol est un échec, le gestionnaire tentera alors de voler à son prochain pair.

3. **Message de détermination** : Le message de détermination est envoyé par un *placeholder* lorsqu'il est déterminé. Il contient les tâches qui étaient suspendues sur ce dernier. Il parvient au gestionnaire dont le travailleur a déterminé le *placeholder*. Il lui communique donc de nouvelles tâches pouvant être exécutées et lui indique du même coup que son travailleur a terminé. Le gestionnaire tentera alors de démarrer une nouvelle tâche. L'interaction entre le travailleur, le placeholder et le gestionnaire est illustrée à la figure 4.8.

4. **Message de terminaison** : Le message de terminaison provient d'un travailleur dont la tâche vient d'être suspendue sur un placeholder non déterminé. Il permet d'indiquer au gestionnaire qu'il peut tenter de démarrer une nouvelle tâche. On peut voir l'illustration des communications entre le travailleur, le placeholder et le gestionnaire à la figure 4.7.

5. **Message de fin de travail** : Le message de fin de travail parvient au gestionnaire ayant démarré le bloc de travail et indique qu'il peut aviser les autres gestionnaires de retourner à leur fil d'exécution respectif. On peut voir une représentation à la figure 4.9.

Un gestionnaire va habituellement tenter de démarrer une nouvelle tâche s'il reçoit un message lui indiquant que son travailleur a terminé, c'est-à-dire un message de

détermination ou de terminaison, ou bien s'il reçoit un message lui indiquant que sa dernière demande de vol de tâche a échoué, c'est-à-dire un message de tâche. Il essaie tout d'abord d'obtenir une tâche de sa *file des tâches lourdes*. Cette file contient des tâches ayant été suspendues à un certain moment, par exemple parce qu'elles ont bloqué sur un *placeholder* et qui sont redevenues exécutables.

Si la file ne contient rien, il enverra alors un message de vol au prochain gestionnaire qu'il doit contacter et se remettra en attente de messages. Une des politiques du système est que si un gestionnaire envoie un message de vol à un autre, il devra contacter tous les autres avant de le recontacter, que le vol ait réussi ou non. Cela améliore la distribution du travail et évite que certains soient beaucoup plus sollicités que d'autres.

Si le gestionnaire effectue des tentatives de vol sans succès à l'ensemble de ses pairs, il devra alors se mettre en attente chronométrée pendant un certain temps afin de ne pas inonder les autres de demandes. Il continuera de répondre aux messages, mais n'essaiera pas de voler de tâches avant un certain temps. La durée de ce temps d'attente est déterminée par l'usager en fonction de l'application. Certaines applications, comme celles ne possédant qu'une seule section parallèle, bénéficieront d'un temps d'attente infini pour éviter d'inonder les travailleurs possédant encore du travail à la fin de cette section.

4.4.2 Vol de tâches légères

Lorsqu'un gestionnaire reçoit une demande de vol et qu'aucune tâche n'est disponible dans sa file de tâches lourdes, il doit alors essayer de créer une nouvelle tâche lourde à partir de la pile d'exécution de son travailleur. Pour cela, il devra interrompre son travailleur et parcourir sa pile afin de vérifier si cela est possible.

Comme le système a pour politique le *vol par le bas*, la pile devra être analysée jusqu'à sa limite inférieure afin de trouver la tâche parente la plus ancienne possible. Lorsqu'elle est trouvée, le travail de création peut commencer.

Une tâche lourde est composée de deux parties distinctes. Tout d'abord un *place-*

holder qui doit recevoir le résultat d'une tâche. Dans le cas du vol par le bas, c'est le résultat de l'enfant de la tâche volée que ce dernier représentera. Ensuite, il y a une continuation réifiée qui doit être créée à partir du frame représentant le début de la tâche, c'est-à-dire le point de retour du `future`.

```
(define (steal-lightweight-task worker)
  (let* ((ph      (make-ph))
         (promise (delay (touch-placeholder ph)))
         (cont    (thread-call
                   worker
                   (lambda ()
                     (break-stack
                      (lambda (result)
                        (ph-determine! ph result)))))))
    (and cont
         (lambda ()
           (continuation-return cont promise)))))
```

La procédure `steal-lightweight-task` commence donc par créer un *placeholder* qui est un processus léger dont la fonction est uniquement de recevoir un résultat et de gérer les tâches en attente de ce résultat. Ce dernier est encapsulé dans une promesse qui consiste uniquement à mettre en attente la demande du résultat au placeholder. Lorsque l'opérateur `touch` est appliqué à une promesse l'opération s'effectue alors à ce moment.

La continuation est créée à l'aide de la procédure `break-stack` qui est exécutée dans le travailleur à l'aide de la méthode `thread-call`. Cette dernière permet d'effectuer un appel de fonction dans le contexte d'un autre thread. Elle est implantée à l'aide de `thread-interrupt!` qui prend un thread et le rend exécutable en le retirant des files d'attente dont il peut faire partie et qui, de plus, lui attache une fonction, une action, à invoquer lorsqu'il sera remis en exécution. On peut voir ici comment `thread-call` utilise `thread-interrupt` :

```
(define (thread-call thread thunk)
  (let ((result-mutex (make-mutex 'thread-call-result)))
    (mutex-lock! result-mutex #f thread)
    (##thread-interrupt!
     thread
```

```
 (lambda ()
   (let ((result (thunk)))
     (mutex-specific-set! result-mutex result)
     (mutex-unlock! result-mutex)
     (##void))))
(mutex-lock! result-mutex #f (current-thread))
(mutex-specific result-mutex)))
```

Il crée tout d'abord un mutex qui servira à contenir le résultat de la fonction à appeler dans le contexte du thread. Puis il barre ce mutex au nom du thread concerné. Il appelle `thread-interrupt!` avec une méthode qui appelle la fonction et insère son résultat dans le mutex avant de le débarrer. Quant au thread initial, après l'appel à `thread-interrupt!`, il tente de barrer le mutex ce qui le fait bloquer jusqu'à ce que l'autre thread l'ait débarré et donc que le résultat soit connu. Ensuite, il prend le résultat dans le mutex et le retourne.

Le paramètre de `break-stack` est une fonction servant à recevoir le résultat de l'enfant de la tâche volée lorsque celui-ci est disponible. On appelle tout simplement la procédure `ph-determine!` afin d'aviser le placeholder de son résultat.

Finalement, on crée la tâche en encapsulant ces deux éléments dans une fermeture qui va invoquer la continuation avec la promesse comme valeur de retour. Le travailleur du gestionnaire ayant volé la tâche se retrouvera donc au point de retour du `future` avec la promesse comme valeur de retour de ce dernier.

4.4.3 Break Stack

La procédure `break-stack` contient la machinerie permettant de parcourir la pile d'exécution du travailleur afin de trouver la continuation de la tâche légère parente la plus ancienne et la transformer en tâche lourde.

Comme il a été vu dans le chapitre 3, l'information concernant un point de retour et son frame de continuation est contenue dans un objet appelé descripteur de point de retour. Il existe un champ supplémentaire dans cette structure de données que nous abordons maintenant. Il s'agit du *type de point de retour*. Dans le système *Gambit-C*, il en existe trois :

1. Points de retour normaux `RETN` : Sont utilisés pour le retour des appels de fonction normaux.

2. Points de retour internes `RETI` : Sont utilisés pour le retour des appels à des routines internes telles que le ramasse-miette où la création d'un frame de continuation normal gaspillerait de l'espace parce que, par exemple, tous les registres doivent être sauvegardés. À ce moment, c'est la routine appelée qui se charge de faire ces sauvegardes.

3. Points de retour de tâches `RETT` : Sont utilisés pour représenter le point de retour d'une tâche enfant dans une tâche parent, c'est-à-dire la continuation d'un `future`.

Donc chaque point de retour d'un `future` est identifié par un frame de continuation ayant un type particulier. Donc pour trouver la tâche parente la plus ancienne il suffit de parcourir la pile et de trouver le frame le plus profond possédant le type tâche (c.-à.-d. `RETT`). La procédure `break-stack` poursuivra aussi sa recherche si les frames de continuations suivants sont présents dans le tas.

Lorsque la routine a trouvé la tâche la plus ancienne, son travail est de séparer la tâche parent, c'est-à-dire la continuation de la tâche enfant, de cette dernière. On devra donc modifier la continuation de la tâche enfant pour une routine qui se chargera de recevoir le résultat de cette dernière et de faire le travail nécessaire, c'est-à-dire le passer à la méthode reçue en paramètre par `break-frame`. Cette dernière se charge habituellement d'indiquer au *placeholder* associé à la tâche parente le résultat de la tâche enfant.

Cette routine se nomme le *end-body handler* et on doit donc chaîner le dernier frame de la tâche enfant avec un frame associé au point de retour situé dans ce *handler*. C'est pourquoi un frame de tâche contient toujours assez d'espace pour pouvoir y insérer un *break frame* chainant vers un frame alloué à cet effet dans le tas.

De plus, la routine doit aussi créer la continuation réifiée qui fera partie de la tâche lourde. Pour cela, on doit créer un objet continuation pointant vers le frame représentant le début de la tâche parent. Toutefois, ce frame ne peut continuer d'être associé

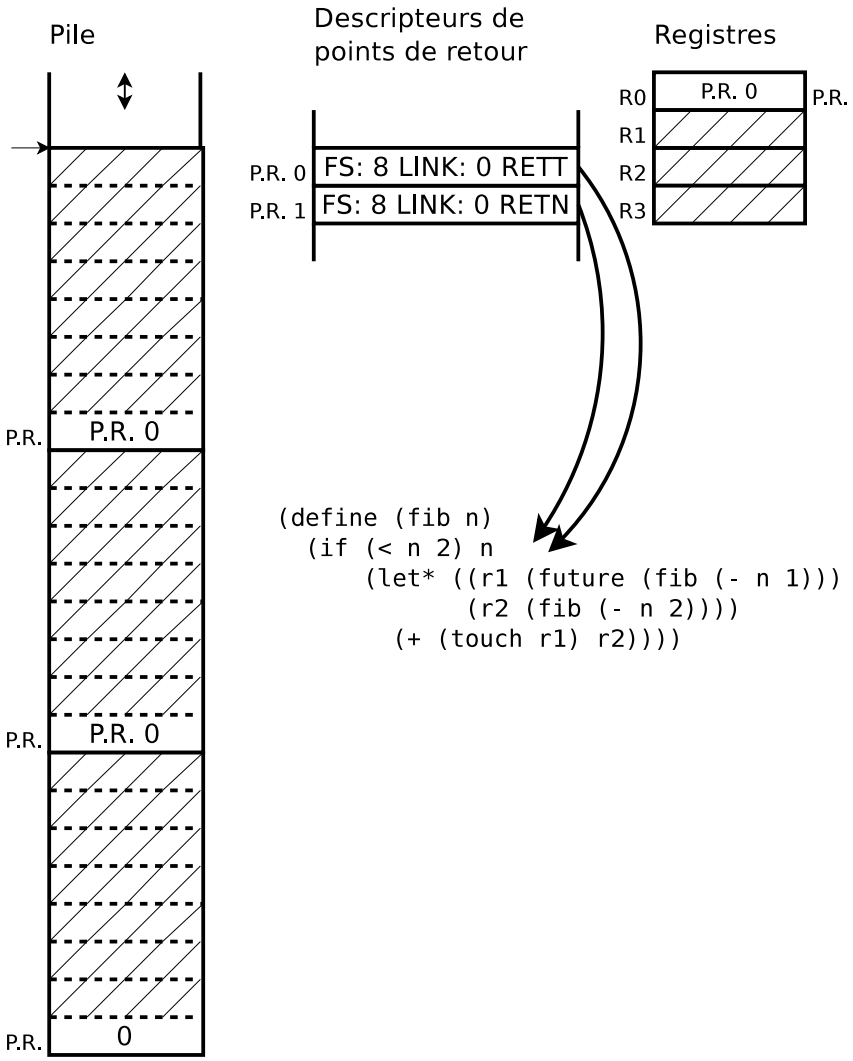

Figure 4.10 – État des structures avant la séparation de la pile

à un descripteur de point de retour indiquant qu'il s'agit d'un retour de tâche, car après la séparation des deux tâches, cela ne sera plus vrai. C'est pourquoi pour chaque descripteur de retour de tâche, il existe un descripteur correspondant de type normal situé à une distance constante. On doit donc changer le champ descripteur de point de retour de ce frame pour ce dernier.

La figure 4.10 montre l'état des structures avant la séparation de la continuation. On peut voir que toutes les frames sont associées à des descripteurs de point de retour `RETT`. Cela n'est pas toujours le cas, mais, par exemple, dans fibonacci, toutes les tâches enfant ont le même point de retour qui est la continuation du `future`.

Dans la figure 4.11, on constate que l'on a modifié l'avant-dernier frame de tâche pour

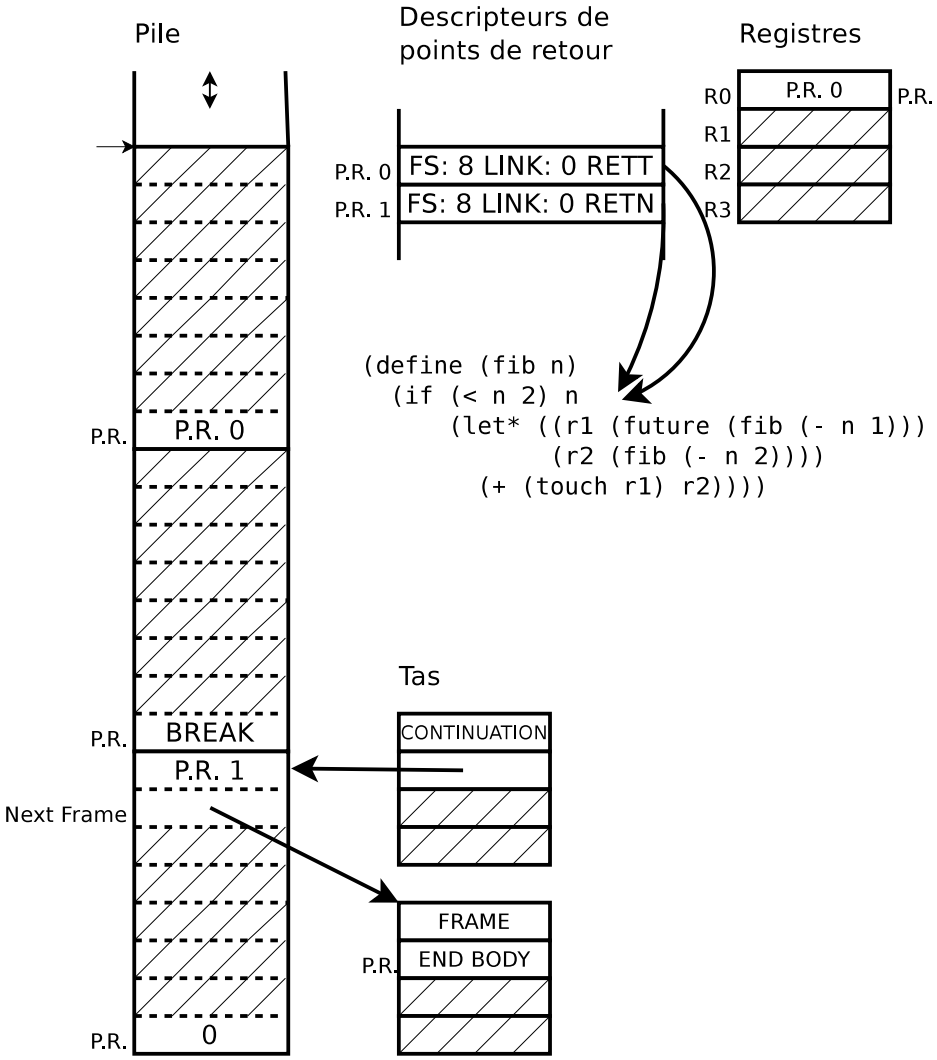

Figure 4.11 – État des structures après la séparation de la pile

indiquer que le prochain frame est maintenant un *break frame*. Ce *break frame* chaîne la pile avec un frame situé sur le tas et qui a comme point de retour le *end body handler*. De plus, on voit que le *break frame* chevauche le premier frame de la continuation de la tâche parente. Enfin, on voit que le premier frame de la continuation réifiée possède maintenant comme descripteur de point de retour un descripteur équivalent, mais de type normal.

4.4.4 Touch des placeholders

Lorsqu'une tâche effectue une opération stricte, par exemple un `touch`, sur la promesse d'une autre tâche, le *placeholder* doit alors être contacté afin d'obtenir le résultat. Il est alors possible que le *placeholder* réponde avec le résultat désiré ou bien qu'il indique que la tâche concernée n'a pas encore terminé.

Si le résultat est disponible, la tâche présente va continuer son travail de manière transparente avec le résultat fourni. Toutefois, si ce dernier n'est pas prêt, elle devra alors se suspendre afin d'attendre le résultat désiré. On appelle cette opération la *suspension sur le placeholder*. La tâche utilisera alors la procédure `touch-undetermined-placeholder`.

```
(define (touch-undetermined-placeholder ph)
  (call/cc
   (lambda (k)
     (let ((task (lambda ()
                   (k (ph-value ph)))))
       (ph-enqueue! ph task)
       (terminate)))))
```

Celle-ci commence par capturer la continuation courante qu'elle va encapsuler dans une fermeture retournant à cette dernière avec le résultat du `placeholder` comme valeur de retour. Lorsque cette fermeture sera exécutée, le contrôle repassera à la sortie de `touch-undetermined-placeholder` en retournant le résultat. Cette dernière est ensuite envoyée au `placeholder` pour être mise dans sa liste des tâches en attente. Lorsqu'il obtiendra son résultat, il se chargera de remettre ces tâches dans une file de

tâches lourdes dans un des gestionnaires de travailleurs. Lorsque tout ceci est complété, la tâche se termine et indique à son gestionnaire qu'il peut se remettre à la recherche de travail.

4.5 Sommaire

L'opérateur `future` de *Multilisp* fournit une solution aux problèmes de MPI. Il permet d'automatiser la distribution du travail entre les noeuds et ainsi de régler le problème de la granularité des tâches. Il aide aussi à simplifier la conception des programmes parallèles en offrant un mécanisme simple pour indiquer facilement les possibilités de parallélisme.

Grâce à la *Création Paresseuse de Tâches*, on peut utiliser les `futures` efficacement sur des applications à granularité très fine, car le coût d'introduction d'une tâche est minime, car le système utilise une représentation très légère pour les tâches tant que ces dernières ne sont pas volées. Lors du vol, la représentation lourde est créée à l'aide de la continuation de la tâche légère.

Les programmes parallèles sont implantés à l'aide d'une librairie de communication permettant à des processus distants de se transmettre explicitement de l'information par passage de messages. Ces outils sont utilisés de manière explicite par le programmeur.

Les programmes sont aussi composés de sections où les `futures` sont utilisés et où le transfert des données et des tâches est automatique. Il n'est nécessaire pour le programmeur que d'indiquer à l'aide de l'opérateur `future` quelles expressions peuvent être parallélisées et le système se charge du reste.

Tous les processus exécutent le même programme et sont démarrés par le gestionnaire global qui se charge de surveiller les conditions d'erreurs et surtout de synchroniser les processus lors du démarrage et de l'arrêt afin d'éviter les conditions de course.

La *Création Paresseuse de Tâches* est implantée à l'aide des continuations de *Gambit-C* et de leur sérialisation. Un thread appelé gestionnaire se charge de surveiller un thread appelé travailleur qui effectue le calcul demandé. Lors d'un vol de tâche, le

gestionnaire va capturer la tâche parente la plus ancienne sous forme d'une continuation et l'envoyer au voleur.

CHAPITRE 5

RÉSULTATS

Dans la conception du système présenté dans ce mémoire, nous visons trois objectifs.

1. Montrer que le vol de tâches est une technique viable de balancement de charge sur les machines à mémoire distribuée. Plus particulièrement montrer que de bonnes performances peuvent être obtenues.

2. Montrer que le parallélisme hiérarchique ainsi que l'usage d'un langage fonctionnel permettent de simplifier la conception des programmes même en lorsqu'on offre un modèle partitionné.

3. Créer un système pouvant servir de base au développement d'un langage offrant un modèle global de parallélisme sur des machines à mémoire distribuée.

Afin de démontrer que ces résultats ont été atteints, nous avons développé un ensemble de programmes de test dans notre langage ainsi que dans deux systèmes destinés aux machines à mémoire distribuée et offrant un modèle partitionné : *MPI* et *Erlang*. Nos programmes nous servent à la fois à comparer les taux d'accélération de notre système ainsi que la taille et la simplicité des programmes.

Dans ce chapitre, nous détaillons tout d'abord l'environnement d'expérimentation dans lequel nos résultats ont été obtenus. Ensuite, nous présentons l'ensemble des programmes que nous avons développés dans les trois systèmes testés. Nous discutons des approches utilisées pour chacun. Enfin, nous présentons les résultats statistiques et les analysons dans la dernière section.

5.1 Environnement d'expérimentation

Pour nos mesures, nous utilisons deux machines différentes. Sur chacune de ces machines, nous avons exécuté les programmes de tests avec un nombre variable de

processeurs utilisés.

La première machine, *Beignet*, est un ordinateur à huit processeurs double-coeurs 64 bits *AMD Opteron 865* cadencés à 1.8 Ghz. Il possède 32G de mémoire vive et chaque processeur contient 1024Ko de mémoire cache. La deuxième machine est une grappe de 8 noeuds, *Clop*, reliés par un réseau *Gigabit Ethernet*. Chaque noeud contient 3.8Go de mémoire vive et un processeur 64 bits *AMD Athlon 64 4000+* cadencé à 2.4 Ghz et possédant 1024Ko de cache.

Tous ces systèmes exécutent le système d'exploitation *Fedora Core* 7 sur lequel nous avons installé et utilisé le compilateur *Gambit-C* 4.6.0, le compilateur *GCC* 4.5.1, la plateforme *OpenMPI* 1.4.2 ainsi que *Erlang* R14B01.

5.2 Programmes

Afin de comparer les trois systèmes, nous avons développé un petit ensemble de programmes de tests qui permettent d'analyser les diverses facettes de ces derniers. Nous nous intéressons tout particulièrement à la performance absolue, aux taux d'accélération ainsi qu'à la taille et à la complexité des programmes.

Ces programmes ne se veulent pas une illustration de conditions réelles d'utilisation. De vraies applications, composées de plusieurs modules et possédant des sections parallèles et des sections séquentielles s'alternant présenteraient des cas d'utilisations intéressants. Elles seraient toutefois moins utiles pour mesurer la performance, car leur niveau d'accélération serait grandement limité par les structures de parallélisme présentes dans les algorithmes. Toutefois, elles tendraient à démontrer les avantages de notre approche pour le développement de multiples modules indépendants introduisant du parallélisme hiérarchique par rapport à la difficulté d'accomplir la même tâche dans *MPI* et *Erlang*. Le balancement de charge dynamique y trouverait encore plus son sens. La sérialisation automatique des données serait aussi un grand avantage dans une telle situation. Par contre, ces programmes plus imposants feraient ressortir le fait que notre système reste un hybride entre le modèle partitionné et le modèle global et montreraient

les faiblesses de l'approche partitionnée pour des applications d'une telle complexité.

La technique utilisée dans tous les programmes de test est celle du *diviser pour régner*. On sépare le problème à résoudre en un nombre de morceaux de même taille et indépendants et chacun des processus accomplit sa partie de manière autonome. À la fin, chaque morceau est récupéré et on les réunit pour former la solution. Dans certains cas, la séparation est faite initialement. Par exemple, on séparera une multiplication de matrices en seize morceaux égaux qui seront distribués à chacun des processus. À ce moment, la réunion des pièces, aussi appelée *réduction* est accomplie d'un seul coup. Dans d'autres cas, la séparation est faite de manière hiérarchique. Par exemple, un processus divise le calcul de fibonacci en deux morceaux, un pour chaque appel récursif. Il en assigne un à un autre processus et en garde un pour lui-même. Les deux processus répètent la séparation en deux dans leurs branches respectives. Dans ce cas, l'étape de *réduction* est faite pour chaque division de travail. Par exemple, le processus qui a assigné le travail à l'autre attend la fin du calcul de la branche et effectue alors la réduction, l'addition des deux résultats.

Comme *MPI* et *Erlang* n'offrent pas de balancement dynamique de charge, il est nécessaire de tenir compte du nombre de processus du système lors de la division du travail. Lorsqu'on effectue celle-ci initialement, il est seulement nécessaire de diviser le travail en un nombre de pièces appropriées sans aucune autre difficulté. Dans les cas où l'algorithme ne se prête qu'à un parallélisme hiérarchique, il est alors nécessaire de concevoir deux versions de la fonction servant à résoudre le problème. La première, la version *parallèle*, contient la logique permettant à la fois d'effectuer les deux branches du calcul ainsi que la réduction, mais aussi celle pour assigner une de ces branches à un autre processus et lui donner la responsabilité de démarrer un sous-ensemble des processus du système. Cette fonction possède donc, en plus des paramètres propres au problème, un ensemble de processus enfants à démarrer et assigne à l'un d'entre eux la moitié du calcul ainsi que la moitié des enfants restants.

Dans *MPI*, les processus sont représentés seulement par un entier indiquant leur

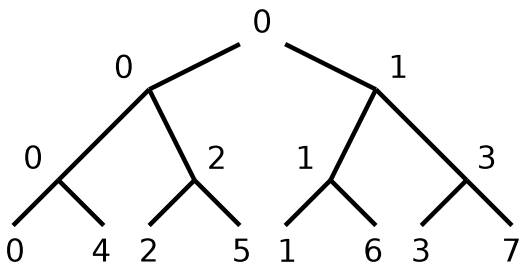

Figure 5.1 – Arbre de démarrage des processus

identification dans le système. L'ensemble des enfants à démarrer est donc implanté à l'aide d'un déplacement par rapport à l'identité du processus courant. Par exemple, au début du calcul, le processus 0 aura pour responsabilité de démarrer les processus 1 à 7. Il contactera le processus 1 et lui indiquera de démarrer 4 processus avec un déplacement de 2. Le processus 1 devra donc démarrer les processus 3,5 et 7 alors que le processus 0 conservera les processus 2, 4 et 6. On voit une illustration de cette manière de démarrer les processus à la figure 5.1.

Dans le cas de *Erlang*, les processus ne sont pas numérotés. Ils ne reçoivent donc qu'un nombre leur indiquant le nombre de processus enfants devant être démarrés pour leur branche du calcul. Comme les processus *Erlang* sont légers, on démarre un enfant pour chaque branche du calcul jusqu'à ce que le nombre à démarrer soit d'un.

Il est certain que cette approche limite le parallélisme dans le cas où le travail dans une branche est plus élevé que dans une autre. Pour remédier à ce problème, nous aurions pu utiliser, dans certains cas, une approche gestionnaire-travailleur mais cela aurait ajouté encore plus de complexité aux programmes *MPI* et *Erlang*.

5.2.1 Fibonacci

Le programme *fibonacci* calcule le nombre du même nom à l'aide de l'algorithme classique à double récursion. C'est un exemple intéressant, car il permet de calculer la performance absolue du système en l'absence d'allocations mémoires et de communication de structures de données entre les processus et dont la granularité des tâches est très petite. De plus, les deux sous-appels ne contiennent pas la même quantité de tra-

vail ce qui est un avantage pour le balancement de charge dynamique, car une division statique du travail causera une perte de temps de calcul sur certains processeurs.

L'implantation en *Multilisp* est très directe. C'est un simple algorithme diviser pour régner où un des deux appels est fait dans un *future*.

Quant à *MPI* et *Erlang*, les algorithmes sont plus complexes, car il est difficile de séparer à l'avance le calcul à faire en un nombre de tâches fixes. C'est pour cette raison qu'une approche diviser pour régner est utilisée où à chaque appel , le processus qui l'effectue va assigner un des deux appels récursifs à un autre processus et effectuer le second lui-même. Lorsque tous les processus devant travailler sont occupés, les appels récursifs subséquents sont effectués avec une version séquentielle de la fonction.

5.2.2 Queens

Le programme *queens* calcule le nombre de manières différentes de placer `n` reines sur un échiquier de taille `n` par `n` sans que celles-ci ne se mettent mutuellement en danger. Il doit donc tenir compte du fait qu'une seule reine à la fois puisse se trouver sur une ligne, une colonne ou une diagonale donnée.

L'implantation multilisp effectue le calcul récursivement sur les lignes. Dans chaque appel récursif, un ensemble de bits représente les colonnes, les diagonales vers la droite et les diagonales vers la gauche qui sont occupées par une reine. Pour chaque position de la ligne qui est libre, on créé un *future* et on explore récursivement les lignes suivante en ajustant les nouvelles positions libres en tenant compte de cette nouvelle reine. L'algorithme est donc diviser pour régner mais n'est pas également balancé.

MPI et *Erlang*, utilisent aussi un algorithme diviser pour régner mais le travail ne peut être séparé de la même façon. En effet, si on utilisait le partitionnement de la version *multilisp*, on devrait assigner la moitié des processus à démarrer à la branche du calcul explorant un seul positionnement de reine sur la ligne et garder l'autre moitié pour explorer le reste de la ligne. La quantité de travail serait alors très différente entre ces deux sous-ensembles du travail. L'autre façon de faire serait d'assigner statiquement

chaque position de la première ligne à un processus, mais on serait alors limité à `N` tâches au maximum. On doit donc se contenter d'une division non optimale où on sépare en deux le nombre de positions à explorer sur une ligne à chaque appel récursif. Cela a pour effet qu'un processus pourrait se retrouver avec un ensemble de positions à explorer qui sont déjà en danger, rendant la quantité de travail à accomplir nulle. On voit donc ici un des avantages du balancement de charge dynamique.

5.2.3 Mandelbrot

Le programme *mandelbrot* calcule pour chaque pixel d'une image si le nombre correspondant fait partie de l'ensemble de mandelbrot. Le résultat de ce traitement donne une image fractale. Pour vérifier si le point fait partie de l'ensemble, on exécute une boucle avec un nombre maximal d'itérations jusqu'à ce que le résultat atteigne une condition donnée ou que ce nombre soit atteint. C'est pour cette raison que chaque point demande une quantité de calcul différente.

La version *Multilisp* utilise une boucle `for` parallèle. Cette dernière prend un intervalle de nombre et une fonction à exécuter pour chacun et sépare, à l'aide de *futures*, cet intervalle de manière diviser pour régner. On boucle donc sur les lignes de l'image résultante et chaque ligne est calculée à l'aide d'une fonction *C* implantée à l'aide du *Foreign-Function Interface* de *Gambit-C*. Nous avons implanté la fonction de calcul en *C* pour éviter certains problèmes d'optimisation du compilateur et pour montrer la possibilité d'utiliser ce langage pour concevoir les parties couteuses du programme. Lorsqu'une ligne est calculée, elle est envoyée au processus 0 qui se chargera de l'entreposer dans un fichier. À la fin du programme, on voit un exemple d'utilisation de principes *programme unique données multiples*. En effet, seul le noeud 0 effectue la réception des lignes et l'entreposage dans le fichier.

Pour la version *MPI* et *Erlang*, on utilise un partitionnement statique déterminé au début de l'exécution du programme. Sachant que les lignes adjacentes représentent une quantité de travail semblable, une ligne est assignée au processus dont l'identification est

égale au numéro de la ligne modulo 2. Donc pour chaque groupe de N lignes adjacentes, chaque processeur en calcule une. Cet algorithme n'est pas optimal et plus le nombre de processus augmente, plus la quantité de travail effectuée par chacun risque de varier. On voit une fois de plus l'utilité de balancement de charge dynamique.

5.2.4 Multiplication de Matrices

Le programme *MM* effectue la multiplication de deux matrices de taille `N` par `N`. C'est un algorithme très intense numériquement et qui fait un grand usage de la mémoire *cache* et du *pipeline* du processeur. Il est nécessaire pour obtenir de bonnes performances de ne pas utiliser l'algorithme classique qui consiste à calculer itérativement chaque point de la matrice résultante, mais plutôt une version se comportant mieux avec la mémoire *cache*. Les versions *Multilisp* et *MPI* utilisent toutes les deux cette approche optimisée.

C'est pourquoi, dans la version *Multilisp*, nous avons programmé la boucle interne avec le langage *C*. On utilise, comme pour *Mandelbrot*, une boucle `for` parallèle qui itère sur les lignes. La ligne résultante est envoyée au noeud 0.

Les versions *MPI* et *Erlang* divisent le travail de manière statique en séparant la première matrice en `N` sous matrices et assignent chacune de ces matrices à un processus. À la fin, toutes ces parties sont réunies pour former la matrice finale. En *Erlang*, il est non seulement impossible d'utiliser la version optimisée de l'algorithme, mais en plus, les matrices doivent être construites à l'aide de structures de données sur lesquelles les mutations sont interdites. On doit donc construire les matrices de manière itérative avec des structures chainées ce qui réduit énormément la performance.

5.3 Analyse

Dans cette section, nous présentons les résultats que nous avons obtenus dans les différents programmes de test et tentons d'expliquer et d'analyser ces derniers. Nous effectuons tout d'abord une analyse générale au niveau des langages et justifions certaines

Programme	TSeq	T16	CExpo	Nombre de cpus				
				1	2	4	8	16
fib 42	12.023	1.467	10.931%	.901	1.658	2.935	5.436	8.197
fib 43	19.425	2.033	8.032%	.926	1.763	3.285	5.76	9.556
fib 44	31.393	2.833	5.975%	.944	1.82	3.462	6.512	11.079
mandelbrot 3072	12.628	1.999	10.823%	.902	1.604	2.715	4.577	6.316
mandelbrot 4096	22.391	2.784	8.21%	.924	1.702	3.011	4.952	8.042
mandelbrot 5120	34.963	4.075	6.835%	.936	1.749	3.176	5.379	8.58
queens 14	1.779	1.17	52.915%	.654	.965	1.05	1.373	1.521
queens 15	10.949	1.62	13.%	.885	1.626	2.59	4.488	6.759
queens 16	73.891	6.406	11.06%	.9	1.777	3.377	6.405	11.534
mm 1536	14.841	8.243	5.346%	.949	1.592	2.31	2.737	1.8
mm 1792	23.499	11.083	.919%	.991	1.699	2.682	3.2	2.12
mm 2048	34.997	15.108	-.263%	1.003	1.751	2.766	3.53	2.316

Tableau 5.I – Table des accélérations sur beignet avec futures

décisions prises lors des mesures. Ensuite, nous étudions les résultats pour chacun des programmes de tests, à la fois sur la machine *Beignet* que sur la grappe *Clop*, et nous tentons d'expliquer leurs particularités.

5.3.1 Analyse Générale

Les tables 5.I, 5.II et 5.III montrent respectivement les taux d'accélération pour les programmes utilisant notre langage, *MPI* et *Erlang* exécutés sur l'ordinateur *beignet*. Les tables 5.IV, 5.V et 5.VI contiennent les résultats des mêmes programmes exécutés sur la grappe *Clop*.

Chaque ligne de la table contient le programme ayant été exécuté ainsi que les mesures suivantes :

1. **TSeq** : Le temps requis, en secondes, pour exécuter la version séquentielle du programme, c'est-à-dire une version équivalente qui ne contient pas de *futures*, de *touch* et aucun appel à des primitives de communication.

2. **T16** : Le temps requis, en secondes, pour exécuter la version parallèle du programme avec 16 processus.

Programme	TSeq	T16	CExpo	Nombre de cpus				
				1	2	4	8	16
fib 45	12.225	3.365	12.849%	.886	1.341	1.943	2.728	3.633
fib 46	19.741	4.469	8.163%	.925	1.43	2.171	3.165	4.417
fib 47	31.936	6.157	5.151%	.951	1.499	2.316	3.526	5.187
mandelbrot 3072	11.601	2.3	13.748%	.879	1.545	2.544	3.675	5.044
mandelbrot 4096	20.612	3.021	7.998%	.926	1.715	2.931	4.794	6.823
mandelbrot 5120	32.482	3.899	4.524%	.957	1.808	3.169	5.504	8.331
queens 14	.799	1.44	203.575%	.329	.4	.457	.521	.555
queens 15	5.031	1.925	39.849%	.715	1.124	1.578	2.118	2.614
queens 16	33.981	4.074	11.986%	.893	1.709	2.918	5.059	8.341
mm 1536	14.055	3.311	11.718%	.895	1.479	2.538	3.608	4.245
mm 1792	22.26	4.297	9.148%	.916	1.614	2.603	4.18	5.18
mm 2048	33.306	5.59	10.532%	.905	1.648	3.078	4.692	5.958

Tableau 5.II – Table des accélérations sur beignet avec MPI

Programme	TSeq	T16	CExpo	Nombre de cpus				
				1	2	4	8	16
fib 42	11.434	2.238	24.577%	.803	1.463	2.291	3.127	5.11
fib 43	18.399	3.468	8.937%	.918	1.431	2.108	3.499	5.305
fib 44	29.647	5.352	15.24%	.868	1.453	2.145	3.347	5.54
mandelbrot 768	15.881	1.436	8.52%	.921	1.605	3.154	5.365	11.062
mandelbrot 896	21.617	1.87	10.725%	.903	1.795	3.107	5.732	11.557
mandelbrot 1024	28.376	2.401	9.696%	.912	1.757	3.2	5.943	11.821
queens 13	1.255	.51	40.344%	.713	1.164	1.389	1.848	2.459
queens 14	6.627	1.082	8.553%	.921	1.591	2.476	3.845	6.127
queens 15	41.943	4.086	.311%	.997	1.72	3.035	5.468	10.265
mm 512	8.593	1.47	11.497%	.897	1.576	2.57	3.836	5.847
mm 640	19.812	2.424	5.123%	.951	1.783	2.937	5.056	8.174
mm 768	35.777	4.159	6.868%	.936	1.772	3.184	5.548	8.603

Tableau 5.III – Table des accélérations sur beignet avec erlang

Programme	TSeq	T16	CExpo	Nombre de cpus				
				1	2	4	8	16
fib 42	9.118	3.039	29.745%	.771	1.213	1.944	2.062	3.
fib 43	14.745	2.905	16.959%	.855	1.443	2.496	3.511	5.076
fib 44	23.788	3.519	12.945%	.885	1.633	2.901	4.23	6.76
mandelbrot 3072	9.578	2.953	29.754%	.771	1.349	2.035	2.271	3.244
mandelbrot 4096	16.923	3.984	14.833%	.871	1.458	2.17	3.251	4.248
mandelbrot 5120	26.424	4.476	16.148%	.861	1.576	2.887	4.055	5.904
queens 14	1.353	2.205	144.984%	.408	.459	.425	.574	.613
queens 15	8.323	3.091	28.542%	.778	1.161	1.767	2.247	2.693
queens 16	56.171	6.786	13.637%	.88	1.652	3.072	5.014	8.278
mm 1536	8.112	8.572	40.892%	.71	1.173	1.44	1.245	.946
mm 1792	12.834	11.245	24.03%	.806	1.311	1.703	1.825	1.141
mm 2048	19.096	14.325	19.712%	.835	1.393	2.	2.097	1.333

Tableau 5.IV – Table des accélérations sur clop avec futures

Programme	TSeq	T16	CExpo	Nombre de cpus				
				1	2	4	8	16
fib 45	9.488	3.288	18.98%	.84	1.263	1.691	2.168	2.886
fib 46	15.35	3.719	13.707%	.879	1.355	2.039	2.825	4.127
fib 47	24.809	5.505	8.692%	.92	1.41	1.98	3.376	4.507
mandelbrot 3072	8.796	5.236	17.561%	.851	1.347	1.933	2.282	1.68
mandelbrot 4096	15.548	5.101	12.963%	.885	1.523	2.399	2.877	3.048
mandelbrot 5120	24.306	4.974	9.726%	.911	1.703	2.723	3.895	4.887
queens 14	.6	1.716	282.962%	.261	.295	.337	.369	.35
queens 15	3.785	2.611	55.701%	.642	.91	1.195	1.158	1.45
queens 16	25.572	4.565	13.413%	.882	1.64	2.808	4.114	5.601
mm 1792	11.504	6.934	14.23%	.875	1.428	1.761	1.663	1.659
mm 1536	7.253	5.708	28.674%	.777	1.295	1.458	1.453	1.271
mm 2048	17.176	8.846	11.482%	.897	1.601	2.022	2.285	1.942

Tableau 5.V – Table des accélérations sur clop avec MPI

Programme	TSeq	T16	CExpo	Nombre de cpus				
				1	2	4	8	16
fib 42	8.478	3.539	23.094%	.812	1.143	1.621	2.107	2.395
fib 43	13.893	3.889	17.084%	.854	1.305	1.95	2.464	3.572
fib 44	22.223	5.332	11.403%	.898	1.412	2.138	2.976	4.168
mandelbrot 768	11.649	3.666	21.162%	.825	1.495	2.494	2.841	3.178
mandelbrot 896	15.792	3.297	16.025%	.862	1.65	2.685	3.416	4.79
mandelbrot 1024	21.343	4.107	10.226%	.907	1.722	2.921	3.93	5.196
queens 15	31.424	5.601	7.293%	.932	1.624	2.79	4.314	5.61
queens 13	.985	2.04	251.922%	.284	.382	.344	.358	.483
queens 14	5.013	2.807	41.166%	.708	1.08	1.444	1.501	1.786
mm 512	5.702	3.396	29.417%	.773	1.127	1.578	1.423	1.679
mm 640	12.866	4.596	16.064%	.862	1.507	2.29	2.591	2.8
mm 768	25.016	4.701	5.411%	.949	1.685	2.96	3.633	5.322

Tableau 5.VI – Table des accélérations sur clop avec erlang

3. **CExpo** : Le coût d'exposition du parallélisme. Il est calculé en divisant le temps d'exécution du programme parallèle (contenant les *futures*, les *touch* et les primitives de communication) par le temps d'exécution du programme séquentiel.

4. **Accélérations** : Pour chaque programme, des exécutions sont effectuées avec un nombre grandissant de processeurs pour mesurer le passage à l'échelle. On obtient l'accélération pour un nombre donné de cpus en divisant le temps séquentiel par le temps obtenu lors de cette exécution. On voit que le temps pour l'exécution parallèle à un processeur provoque un ralentissement à cause du coût d'exposition.

Il est important d'indiquer que pour augmenter la fiabilité des résultats, nous avons tenté d'éliminer les éléments pouvant causer des anomalies. Le seul qui avait réellement une influence était le temps de démarrage des processus. Il arrivait parfois que la courbe des taux d'accélérations ait une forme inattendue. Par exemple, l'accélération à 4 processus était meilleure que celle à 8, mais moins bonne que celle à 16. En investiguant nous avons découvert que le démarrage des processus, autant local que distant causait ces problèmes. C'est pourquoi dans tous nos tests, nous démarrons 16 processus système même si le test en utilise un nombre moins grand. La seule exception à cette règle est *Erlang* sur *Beignet* car il n'est nécessaire de démarrer qu'un seul processus système

	Nombre de processus				
Système	1	2	4	8	16
futures	.476	.491	.508	.453	.559
mpi	.127	.872	.151	1.182	1.295

Tableau 5.VII – Table des temps de démarrage en secondes sur Beignet

	Nombre de processus				
Système	1	2	4	8	16
futures	.951	1.115	1.678	2.145	2.214
mpi	.094	1.231	1.183	1.597	2.438
erlang	1.074	1.081	1.229	1.956	1.962

Tableau 5.VIII – Table des temps de démarrage en secondes sur Clop

grâce aux processus légers. Afin de garder en tête le coût que ce choix implique sur les performances relatives, et surtout sur le coût d'exposition, nous avons inclus les tables 5.VII et 5.VIII qui présentent le temps nécessaire pour charger les processus systèmes sur chaque machine.

Les versions séquentielles des programmes *MPI* sont systématiquement plus performantes que les versions équivalentes en *Multilisp* et en *Erlang*. Dans les programmes où nous avons seulement utilisé seulement du *Multilisp* les performances présentent des différences marquées. Par contre, là où nous avons utilisé le langage *C* pour les boucles internes, les performances séquentielles présentent de petites variations. Cela est dû à la plus grande difficulté d'optimiser un langage dynamique comme *Scheme* et donc résulte en des pertes de performance dans des applications à caractère numérique intensives comme celles présentées ici. Cet effet serait beaucoup moins prononcé dans des applications plus grandes et de nature plus symbolique comme un compilateur.

Au niveau des taux d'accélération, le système *Multilisp* présente pour des programmes de tests ayant des temps séquentiels comparables, de meilleures accélérations que *MPI*. De plus, plus le temps total de calcul est faible, plus cette différence est prononcée. En règle générale, plus la taille de la tâche à effectuer est grande, plus les taux d'accélération seront grands. Cela s'explique par le fait que la taille des sous-tâches augmente en conséquence et permet aux processeurs d'accomplir plus de travail parallèle

	Nombre de cpus			
Programme	2	4	8	16
fib 42	.125	.364	1.056	2.64
fib 43	.129	.359	1.007	3.031
fib 44	.132	.362	1.181	2.963
mandelbrot 3072	5.165	8.723	11.16	13.259
mandelbrot 4096	8.692	14.551	18.356	21.201
mandelbrot 5120	13.278	22.065	27.318	31.011
queens 14	.152	.499	1.251	2.866
queens 15	.179	.676	1.459	3.805
queens 16	.167	.81	2.164	5.132
mm 1792	32.729	87.743	193.062	401.07
mm 2048	42.353	114.183	251.856	522.915
mm 1536	23.96	64.529	142.364	295.352

Tableau 5.IX – Table du volume de données transféré par les futures

sans avoir besoin de communiquer et de voler des tâches. Cela permet aussi d'amortir les coûts séquentiels de préparation tels que le démarrage des processus et les temps d'attente relatifs au transfert des données. La multiplication de matrices, en *Multilisp*, souffre grandement du problème de transfert ce qui explique les performances diminuant à 16 processus. Nous en analysons les raisons plus en détail dans la la section de ce programme.

La table 5.IX permet de comparer le volume de données transféré lors de l'exécution des programmes en *Multilisp*. Cet élément a une grande influence sur les performances de ces derniers et il est donc important de le garder en tête. Comme le volume de données transféré était pratiquement égal entre les exécutions sur *Beignet* et *Clop*, nous n'avons inclus qu'une seule table. Nous avons aussi inclus les graphiques 5.2, 5.3, 5.4 et 5.5 qui présentent la même information.

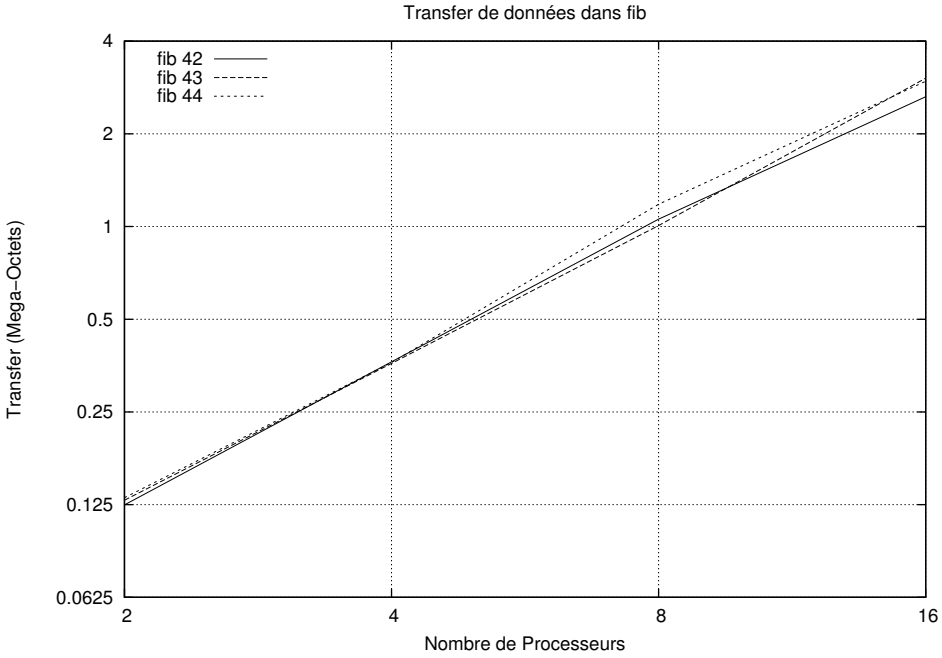

Figure 5.2 – Volume de transfert de fibonacci

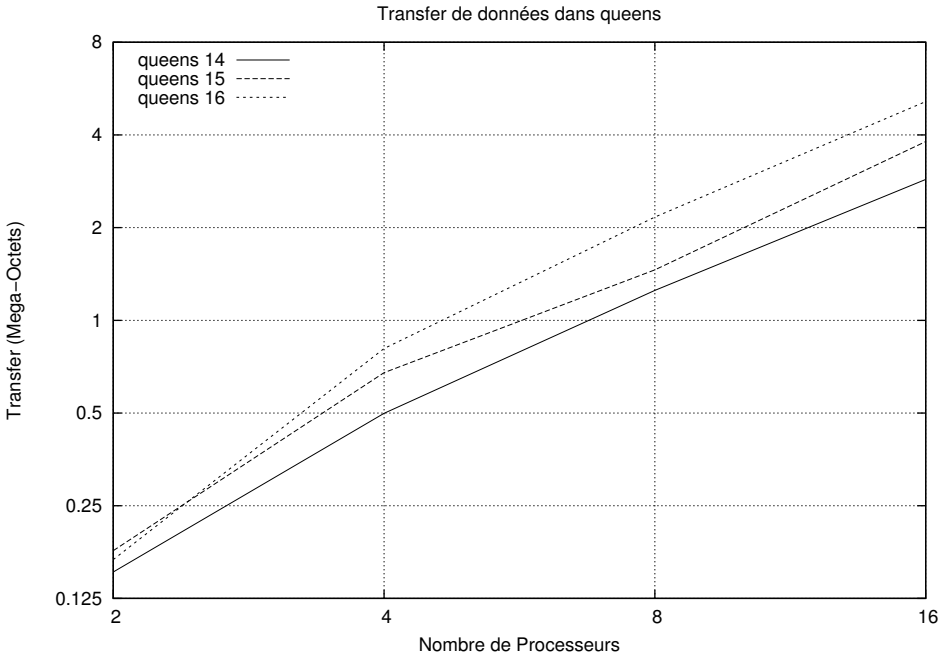

Figure 5.3 – Volume de transfert de queens

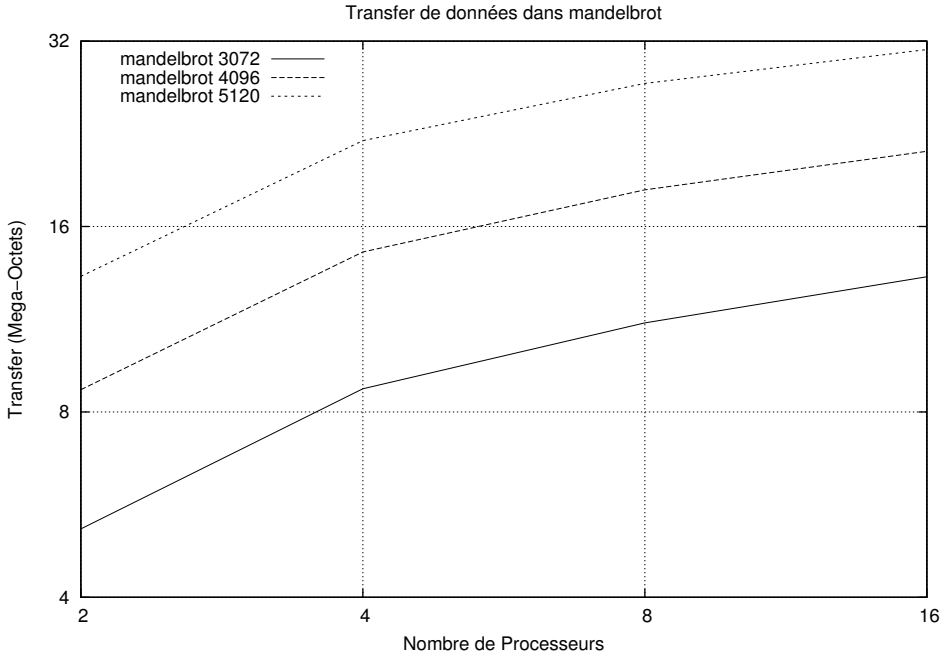

Figure 5.4 – Volume de transfert de mandelbrot

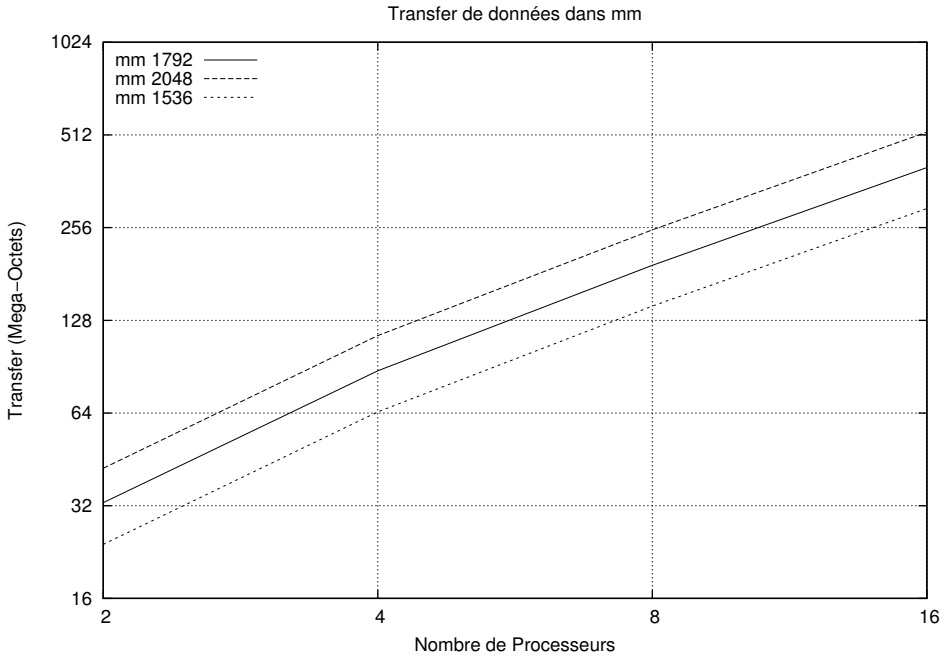

Figure 5.5 – Volume de transfert de la multiplication de matrices

5.3.2 Fibonacci

Dans ce programme, *MPI* surpasse de loin les deux autres langages en terme de performances absolues. Il obtient la même vitesse séquentielle, 12 secondes, sur `fib(45)` que *Erlang* et *Multilisp* sur `fib(42)`. Comme `fib(45)` représente, à toute fin pratique, la même quantité de travail que `fib(42)` et `fib(43)`, qui prennent respectivement 19 et 31 secondes pour *Erlang* et *Multilisp*, dans ce programme *MPI* obtient donc des performances séquentielles environ quatre fois supérieures aux deux autres.

Par contre, au niveau des taux d'accélération, notre système obtient de bien meilleures accélérations, grâce au partage de charge dynamique. En effet, comme les deux branches du calcul de fibonacci ne sont pas balancées, le partitionnement effectué dans la version *MPI* et *Erlang* limite le taux d'accélération. Le *chemin critique*, c'est-à-dire la longueur de la plus longue branche parallèle du calcul pour un certain partitionnement, passe toujours par la branche la plus lourde de l'algorithme et cette dernière contient environ 62% du travail à effectuer. Il est donc possible de calculer la proportion du travail qui sera effectué dans le *chemin critique* avec l'équation $0.62^{log(P)}$ où P est le nombre de processus utilisé. Avec 16 processus, on obtient 14.78% du travail effectué dans le *chemin critique* et donc un taux d'accélération théoriquement limitée à 6.77. Il s'agit, bien entendu, d'une simplification de la quantité de travail effectué dans le *chemin critique*, car il n'inclut que les calculs effectués à partir du moment où chaque processus a reçu sa tâche, c'est-à-dire l'ensemble des cinquièmes appels récursifs lorsqu'on utilise 16 processus. Il exclut aussi tout le temps de démarrage de ces derniers ainsi que toute la gestion supplémentaire nécessaire à la distribution du travail. Nous pouvons donc conclure que le taux maximal d'accélération sera en réalité plus bas que 6.77.

La version *MPI* du programme obtient, comme meilleur taux, une accélération de 5.19 sur `fib(47)` qui prend 32 secondes en version séquentielle. Quant à *Erlang*, on voit que son taux d'accélération à 16 processus augmente lentement de 5.11 à 5.54 sur des programmes aillant une grande différence de temps d'exécution. Cela semble indiquer qu'il s'approche de la limite réelle d'accélération. En *Multilisp*, comme la taille du

chemin critique n'est pas fixée au démarrage, mais bien dynamique grâce au partage de charge, on voit un tout autre comportement. On obtient un taux d'accélération de 11.1 sur `fib(44)`, et sur un test effectué rapidement, nous avons testé `fib(45)` en *Multilisp* avec 16 processus et avons obtenu un temps d'environ 4.5 secondes par rapport aux 3.3 secondes de *MPI*.

Sur la grappe *Clop*, les accélérations sont beaucoup moins bonnes, ce qui s'explique par des coûts d'exposition plus grands à cause du démarrage des processus distants ainsi que de la plus grande vitesse des noeuds ce qui accélère le calcul et empêche de bien amortir coûts. Comme nous avons expliqué précédemment, afin d'éliminer certaines anomalies des résultats de nos tests, nous démarrons 16 processus, peu importe le nombre de processus que le calcul utilisera. Sur la grappe *Clop* cela représente un temps d'environ 2 secondes qui contribue à réduire l'accélération relative. Avec des tests plus longs qui amortiraient ce coût, des taux d'accélérations comparables à ceux de *Beignet* seraient possibles.

On peut voir sur le graphique 5.2 que la quantité totale de données transférée n'a pas tendance à augmenter selon la taille de la tâche à effectuer, mais seulement en fonction du nombre de processus ce qui permet à ce programme de bien passer à l'échelle. Cela indique que c'est le nombre de tâches supplémentaires transférées qui est à l'origine de cette augmentation.

Il faut noter que la version *Multilisp* séquentielle aurait pu être accélérée d'un facteur de deux si nous avions permis au compilateur d'effectuer l'inlining des fonctions, mais nous avons empêché cette optimisation, car elle nuisait à la version parallèle et indiquait un énorme coût d'exposition dont nous voulions faire abstraction.

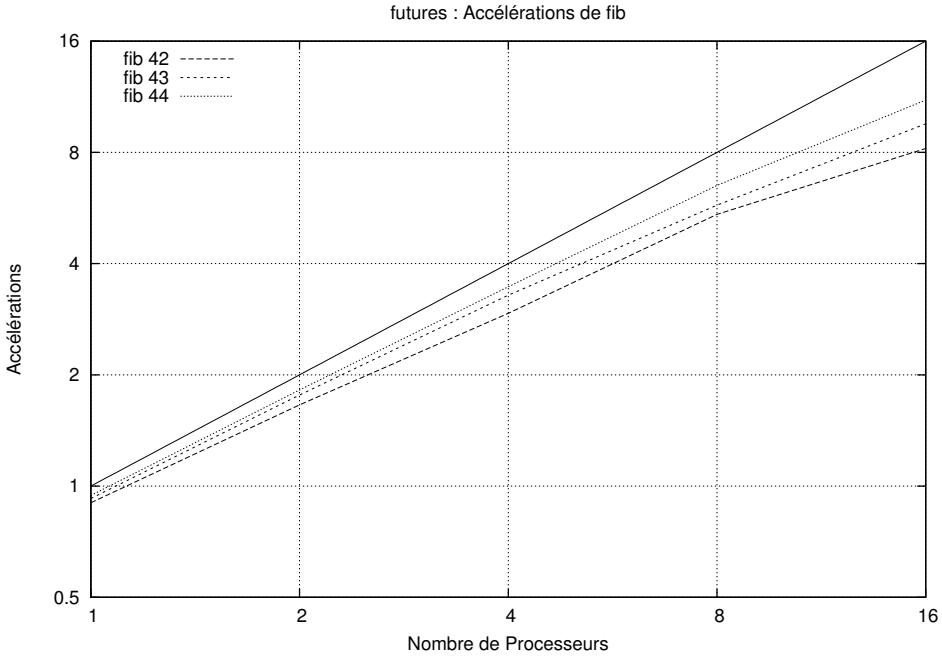

Figure 5.6 – Accélérations de fibonacci avec futures sur beignet

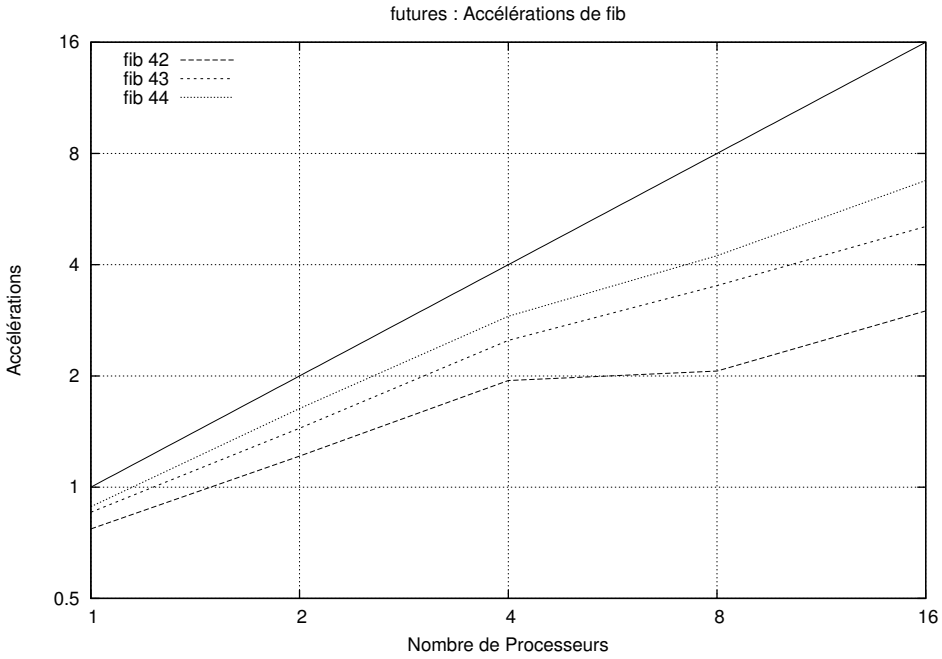

Figure 5.7 – Accélérations de fibonacci avec futures sur clop

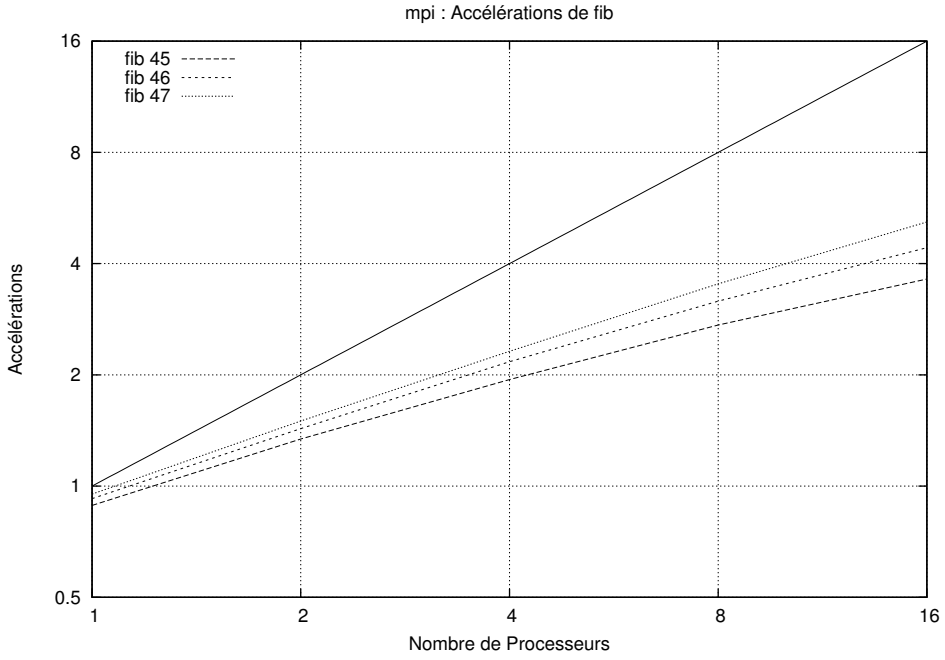

Figure 5.8 – Accélérations de fibonacci avec MPI sur beignet

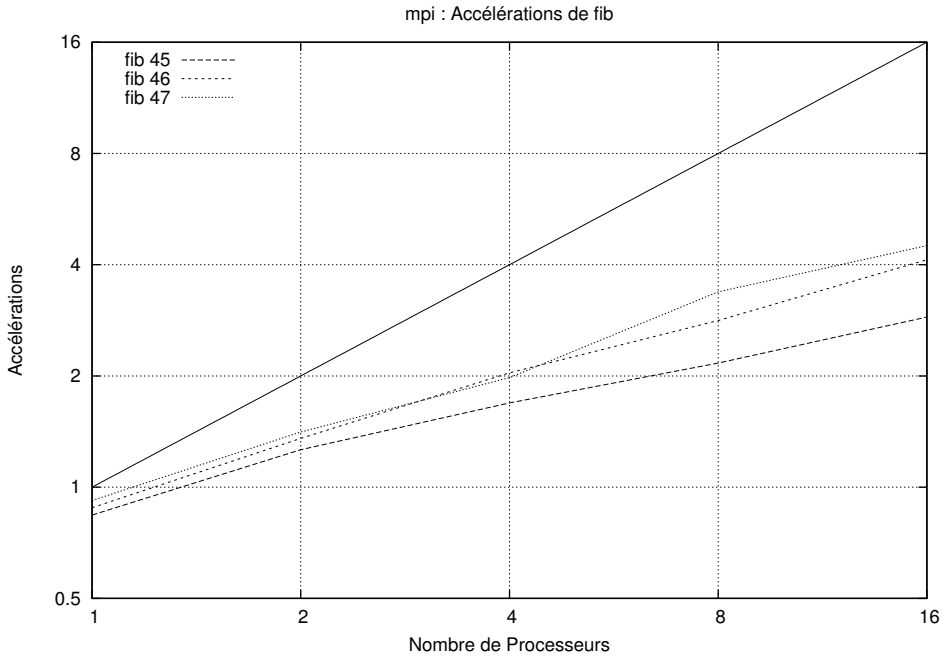

Figure 5.9 – Accélérations de fibonacci avec MPI sur clop

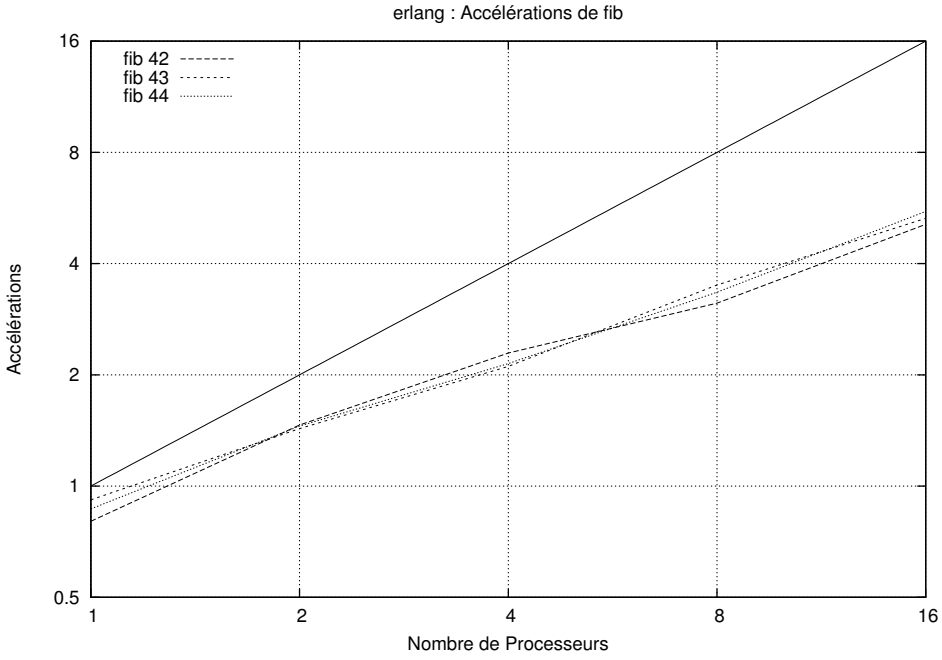

Figure 5.10 – Accélérations de fibonacci avec Erlang sur beignet

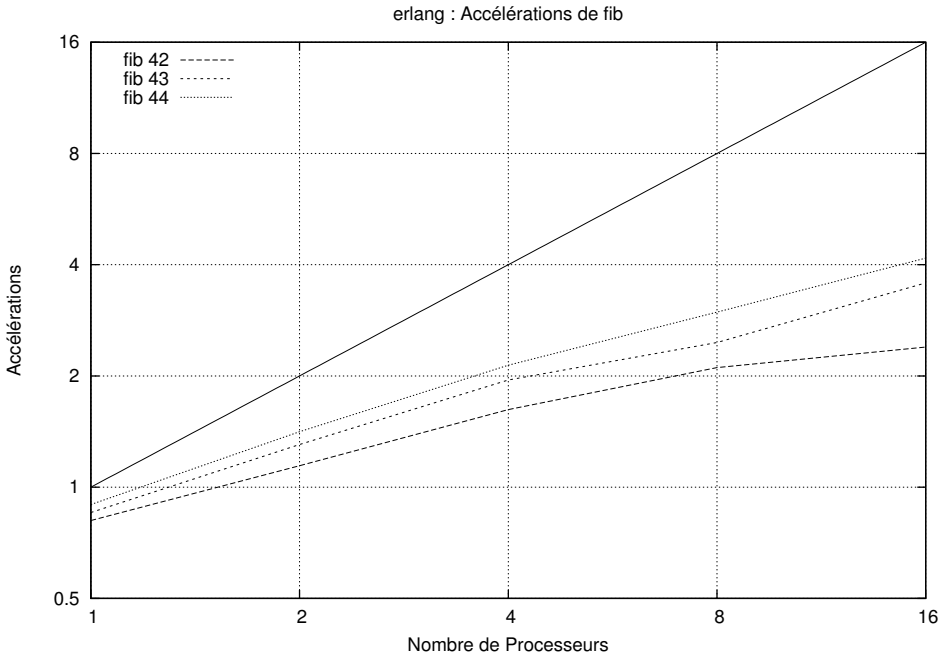

Figure 5.11 – Accélérations de fibonacci avec Erlang sur clop

5.3.3 Queens

Le programme *queens* se compare à *fibonacci* en terme de potentiel d'accélération. Dans ce programme, la version séquentielle en *MPI* est encore la plus performante. Toutefois, la version *Multilisp* reste environ 2 fois moins rapide, ce qui est une amélioration par rapport à fibonacci. La raison est que nous n'avons pas empêché l'inlining du compilateur dans le programme séquentiel, car la différence était beaucoup moins prononcée qu'avec *fibonacci*. C'est aussi ce qui explique le coût d'exposition restant élevé dans `queens(16)` malgré le fait que les coûts de démarrage devraient être bien amortis dans une exécution durant environ 73 secondes. La version *Erlang* présente des temps séquentiels bien moins performants que ceux des deux autres langages.

Les taux d'accélérations, pour *Multilisp*, restent toutefois moins bons que ceux obtenus pour *fibonacci*. Pour un test durant 11 secondes, on obtient un taux d'accélération de 6.8 alors qu'un test de 12 secondes nous permettait un taux de 8.2 avec *fibonacci*. Le coût d'exposition explique en partie cet effet. Une autre raison est que le partitionnement du travail dans la version *Multilisp* de *Queens* ne se fait pas de manière égale. Une des branches possède beaucoup plus de travail que l'autre. Cela peut avoir pour effet d'occasionner un plus grand nombre de vols de tâches ainsi qu'un plus grand nombre d'attentes pour des résultats. Ces couts sont amortissables avec une plus grande taille de programme et c'est pourquoi on atteint un taux d'accélération de 11.5 à 16 processus sur un programme durant 73.8 secondes.

En ce qui concerne *MPI*, on remarque tout d'abord le coût d'exposition qui semble élevé. Cela s'explique par le fait que la version séquentielle de la fonction qui est présente dans la version parallèle du programme, celle que les processus exécutent lorsqu'il n'y a plus de tâches à démarrer, n'est pas exactement la même que dans la version séquentielle du programme. Il a en effet fallu ajouter deux paramètres pour gérer le cas où un processus ne démarre pas son travail à la première colonne, un cas qui ne peut exister dans la version séquentielle. C'est deux paramètres supplémentaires occasionnent sur une exécution de 34 secondes un coût d'environ 2 secondes, donc un coût d'exposition

supplémentaire d'environ 5.8%.

Au niveau des accélérations, *MPI* ne dépasse pas 8.3. Cela est probablement dû à l'augmentation de la taille du chemin critique à cause des instructions supplémentaires occasionnées par la version différente de la fonction. Il est très difficile de calculer l'accélération maximale théorique dans ce cas, mais si on considère le cas de *Erlang*, où la fonction séquentielle est la même dans les deux versions du programme, on voit que sur un programme durant 42 secondes, il obtient une accélération de 10.2, ce qui laisse penser que la version *MPI* n'est pas limitée par un problème dans l'algorithme.

Sur clop, les accélérations sont moins bonnes. Une fois de plus, cela est dû au temps de démarrage plus long et à la latence des communications qui augmente la taille du chemin critique. Le volume de données transférées est minime comme dans le cas de fibonacci.

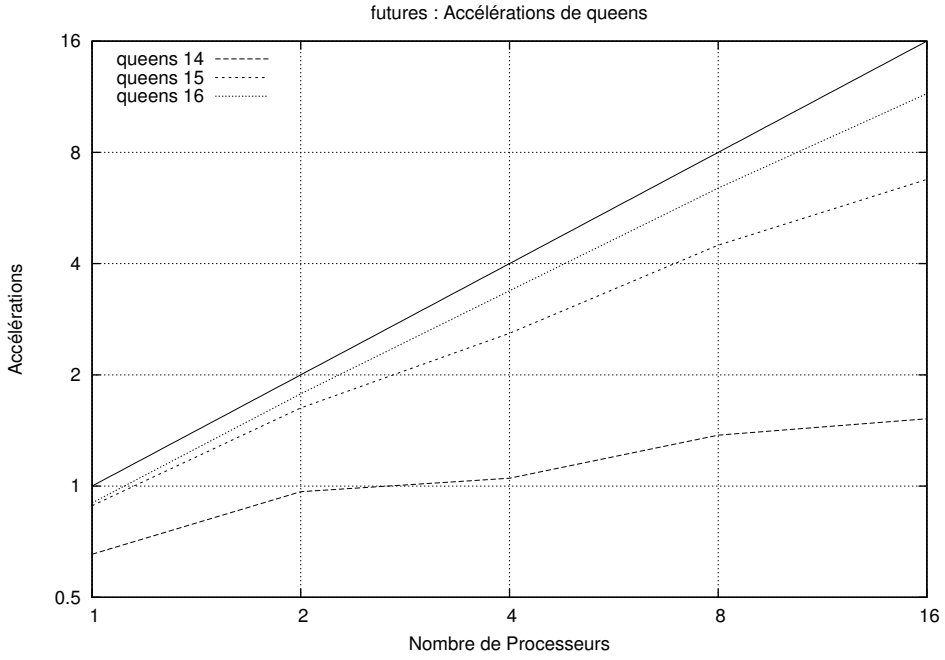

Figure 5.12 – Accélérations de queens avec futures sur beignet

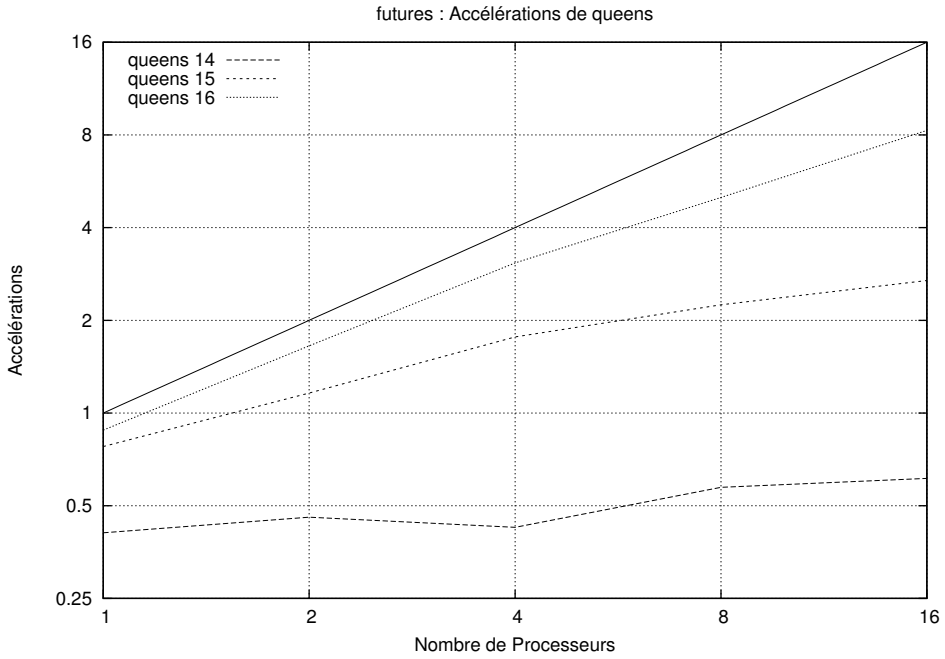

Figure 5.13 – Accélérations de queens avec futures sur clop

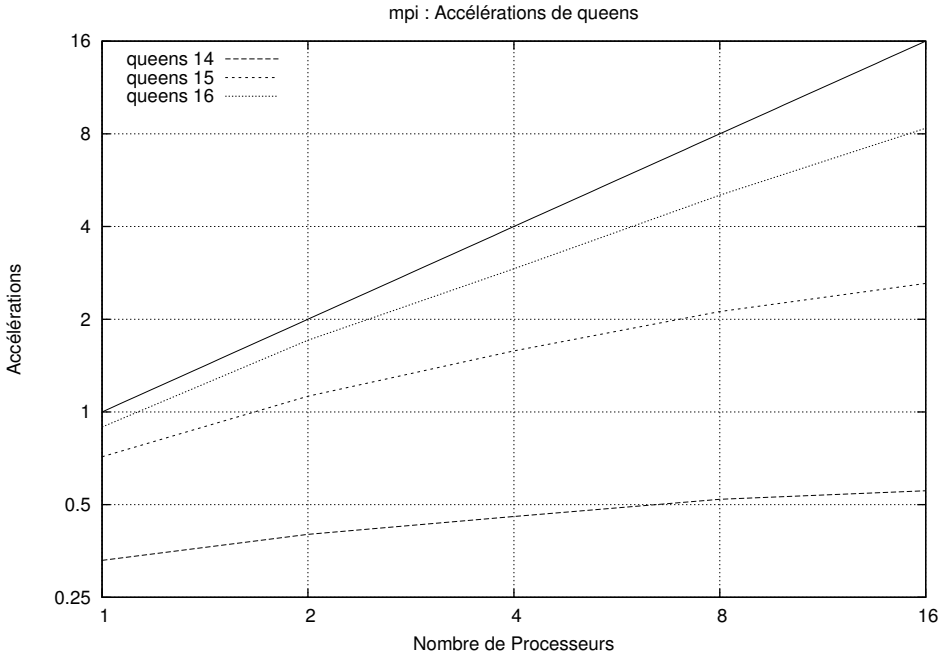

Figure 5.14 – Accélérations de queens avec MPI sur beignet

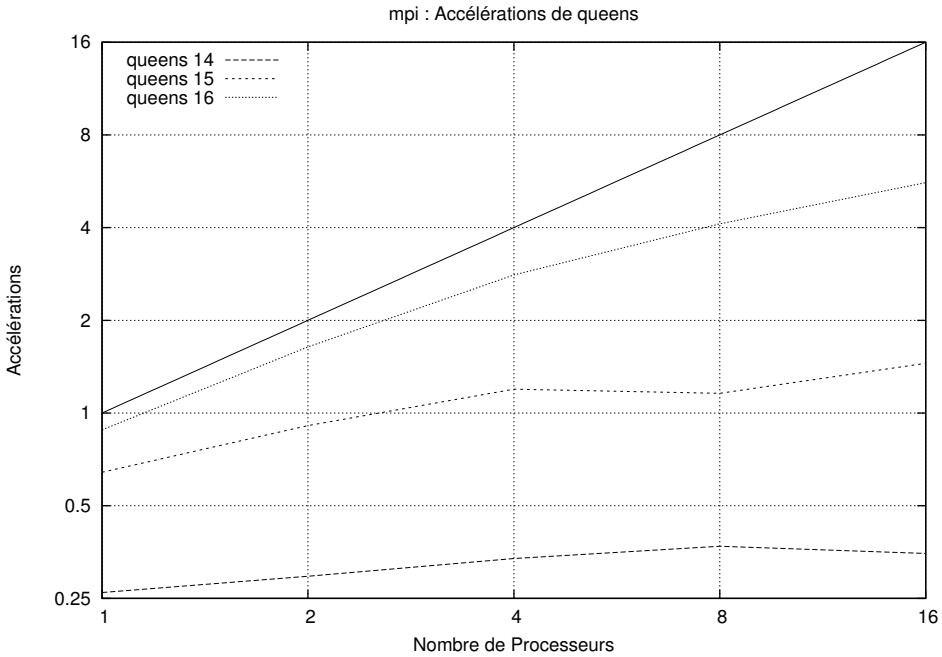

Figure 5.15 – Accélérations de queens avec MPI sur clop

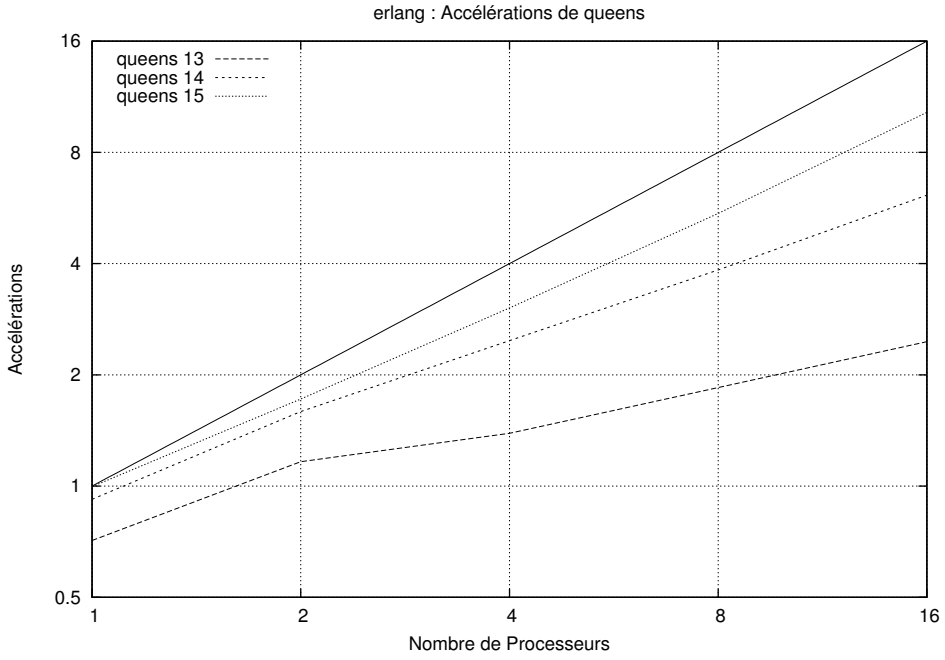

Figure 5.16 – Accélérations de queens avec Erlang sur beignet

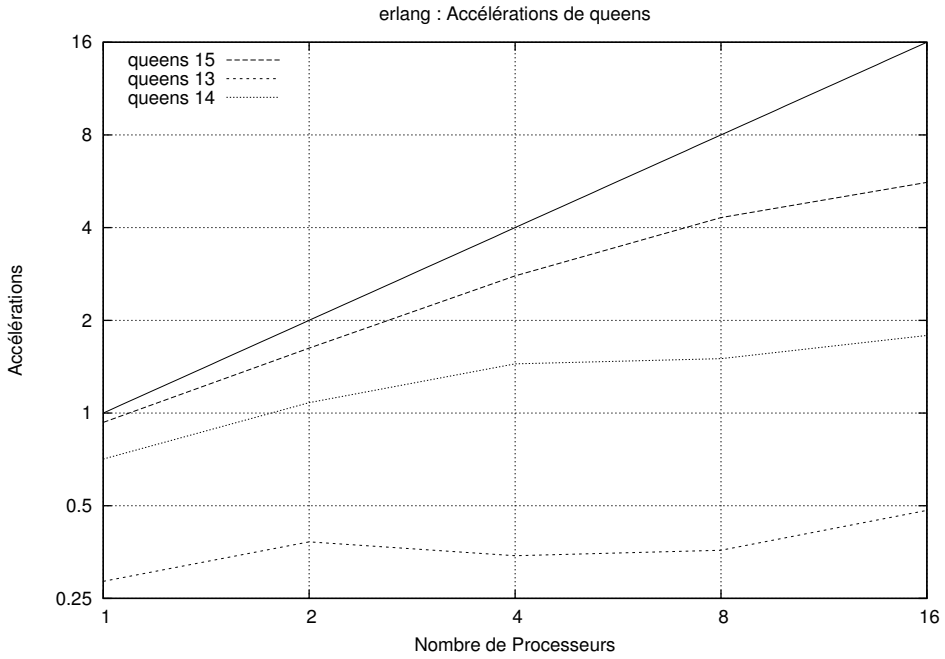

Figure 5.17 – Accélérations de queens avec Erlang sur clop

5.3.4 Mandelbrot

Dans le programme *Mandelbrot*, les temps séquentiels de *Multilisp* et *MPI* sont environ équivalents, car la boucle internet utilise le langage *C* dans la version Multilisp. La version *Erlang* ne peut se comparer, car la matrice résultante doit être construite à l'aide de structures de données sur lesquelles les mutations sont interdites.

Les accélérations de *Erlang* sont très bonnes, entre 11 et 12 pour les trois tests effectués. Cela s'explique par le fait que pour chaque partie du problème, le programme *Erlang* effectue beaucoup plus de travail et transmet moins de données. En fait, les bons résultats d'accélération sont directement dûs aux mauvaises performances de ce dernier sur ce programme. En ce qui concerne *MPI* et *Multilisp*, ils obtiennent tous les deux environ 8.5 de facteur d'accélération sur le plus gros problème qui durent environ 32 secondes. Dans le cas de *MPI* c'est un bon résultat et s'explique par le fait que le chemin critique n'est pas beaucoup plus élevé que les autres chemins dans ce programme.

Par contre, le transfert de la matrice, à la fin du programme, occasionne des coûts aux deux systèmes par rapport à leur version séquentielle. C'est pourquoi l'accélération semble limitée par rapport à *fibonacci* et *queens* pour la version *multilisp* alors que ce problème se parallélise aussi bien. Sur *Clop* ce problème est encore plus important, car la matrice doit être transférée sur le réseau ce qui augmente encore plus la taille du chemin critique et occasionne des accélérations réduites.

Mandelbrot est un exemple de programme qui bénéficierait d'opérations propres au langage pour effectuer le parallélisme de données. Si la matrice pouvait être distribuée sur les processus, il serait possible, en *Multilisp*, d'obtenir une distribution du travail exactement comme celle des versions *MPI* et *Erlang* et qui est relativement optimale vu les accélérations possibles dans *Erlang*. C'est un problème où la distribution pseudoaléatoire du travail du vol de tâches n'est pas un atout, mais où une distribution automatique basée sur la localisation des données serait appropriée.

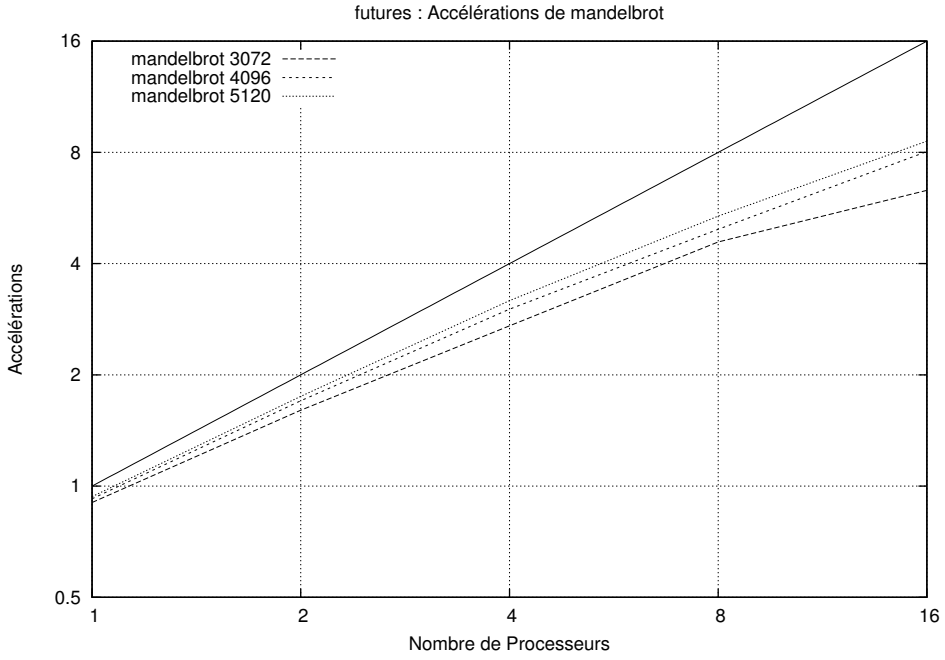

Figure 5.18 – Accélérations de mandelbrot avec futures sur beignet

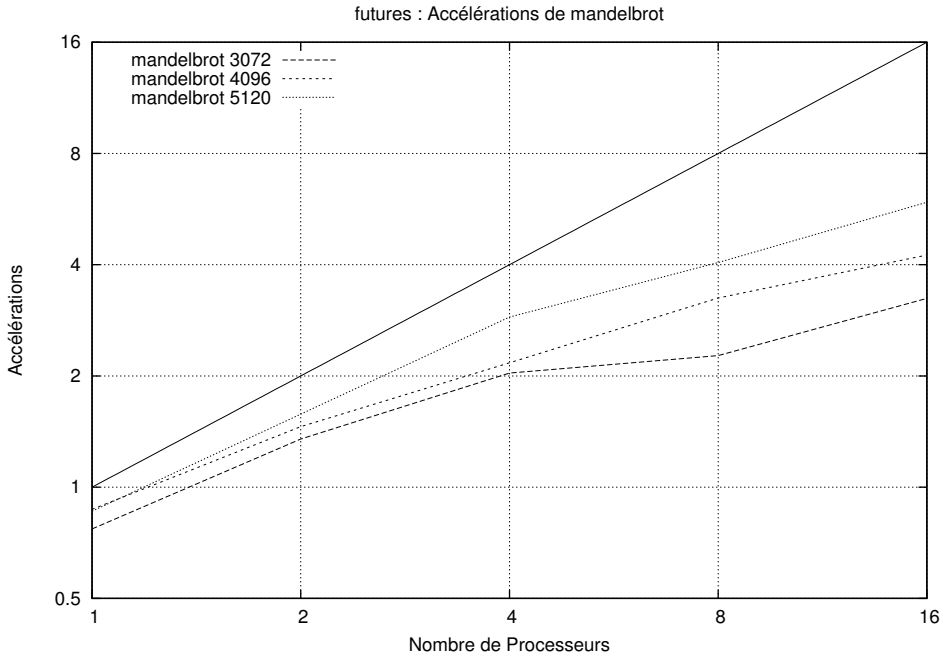

Figure 5.19 – Accélérations de mandelbrot avec futures sur clop

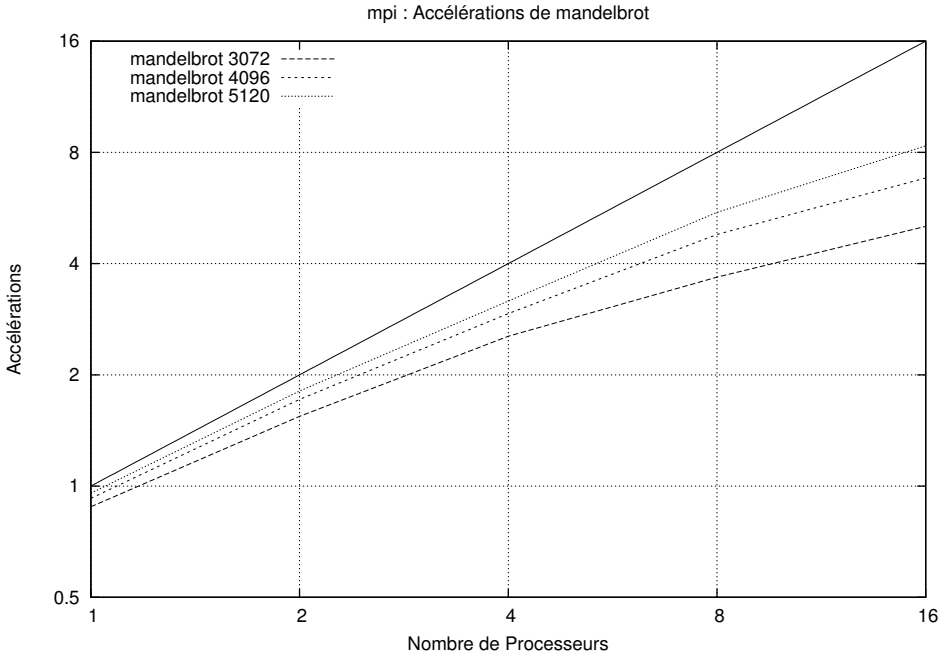

Figure 5.20 – Accélérations de mandelbrot avec MPI sur beignet

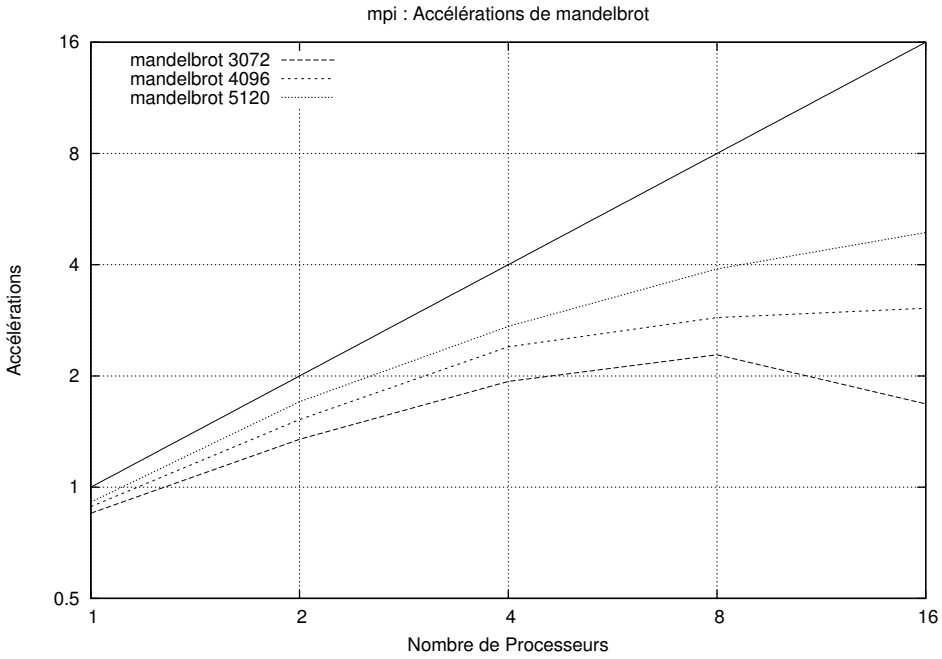

Figure 5.21 – Accélérations de mandelbrot avec MPI sur clop

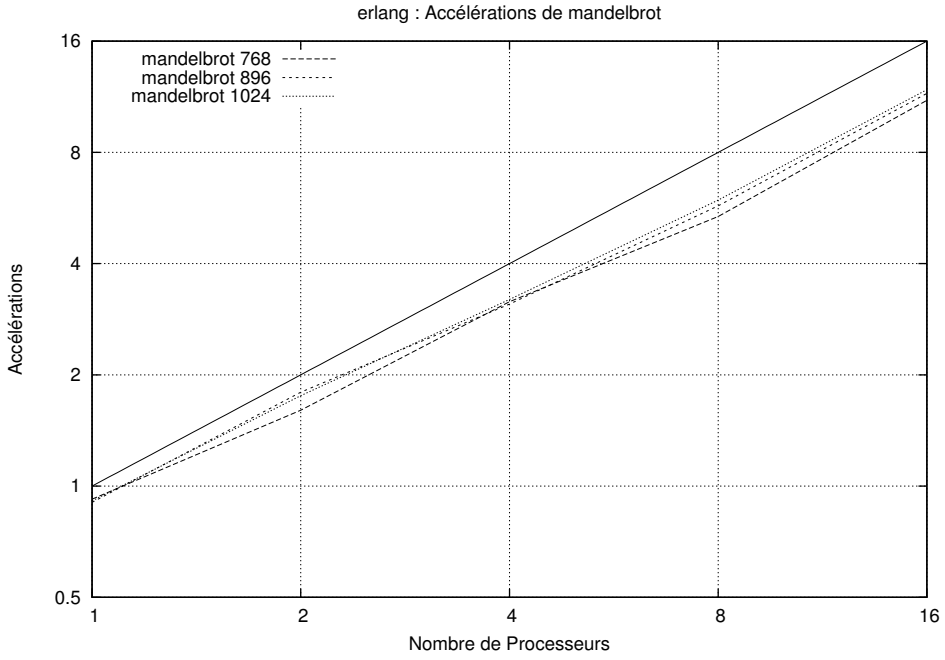

Figure 5.22 – Accélérations de mandelbrot avec Erlang sur beignet

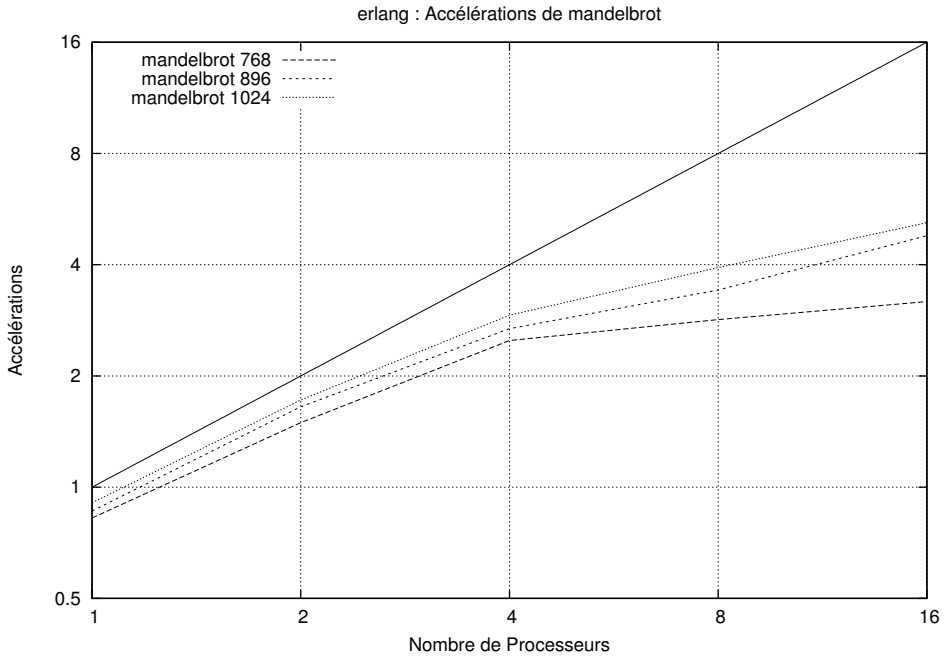

Figure 5.23 – Accélérations de mandelbrot avec Erlang sur clop

5.3.5 Multiplication de matrices

Comme pour *Mandelbrot*, la multiplication de matrice en *Multilisp* utilise le langage *C* pour l'implantation de ses boucles internes. Les performances séquentielles sont donc comparables. Une fois de plus la version *Erlang* ne peut se comparer aux deux autres versions.

Le programme de multiplication de matrice est l'endroit où les accélérations de *Multilisp* sont les moins bonnes et cela démontre une des faiblesses du vol de tâches. Comme on ne sait pas où chaque tâche sera exécutée, il est nécessaire de transférer les deux matrices en entier vers tous les processus du système. Donc, ce coût augmente en fonction du nombre de processus utilisé. On voit donc comme résultat que l'accélération à 16 processus est moins bonne qu'à 8 sur *Beignet*. L'effet sur *Clop* est encore plus prononcé.

MPI s'en sort beaucoup mieux, car il sépare le travail à l'avance et n'a qu'à transférer une partie de la première matrice, les lignes que le processus utilisera. Nous aurions pu faire un algorithme encore plus efficace qui aurait séparé le travail en bloc. À ce moment, on aurait eu qu'à transférer les lignes et les colonnes requises et il aurait été possible de configurer les colonnes en mémoire pour qu'elles agissent bien avec la mémoire cache. Les accélérations de *MPI* se rendent à 6 pour une multiplication durant environ 33 secondes ce qui reste une performance relativement médiocre pour un calcul qui se parallélise aussi facilement. Les accélérations de *Erlang* sont encore une fois excellentes à cause de ses performances médiocres sur le problème.

Enfin, il y a une anomalie dans les résultats de *Multilisp* sur *Beignet*, un coût d'exposition négatif pour `mm(2048)`. Cela est dû à certaines défaillances des processeurs sur cette machine. Nous avons fait plusieurs tests qui démontrent que des pics de performance sont occasionnés de manière aléatoire dans les applications faisant beaucoup usage de la mémoire cache en lecture. C'est pourquoi ce problème ne ressort que dans ce programme.

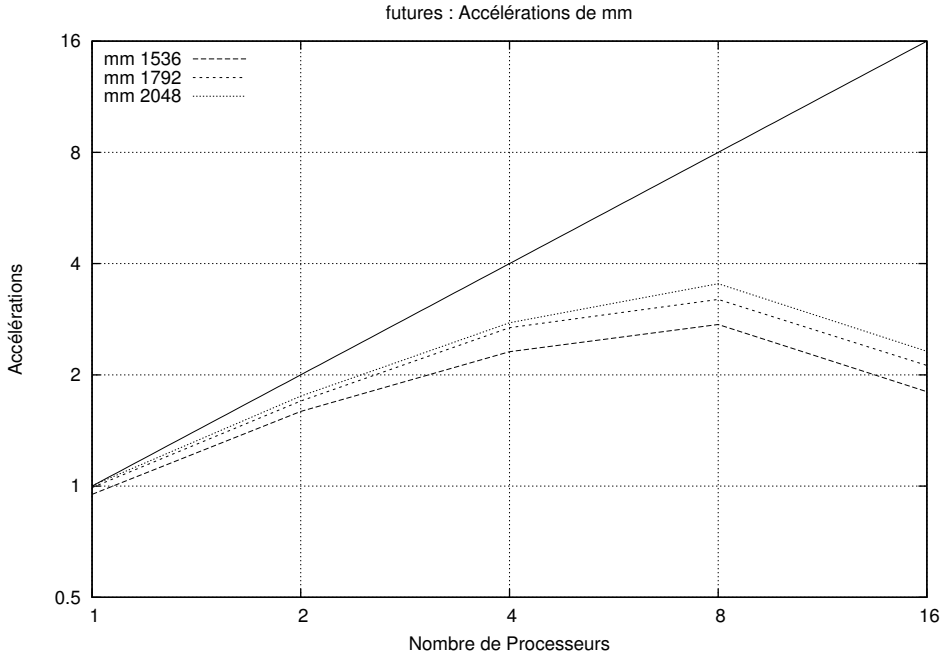

Figure 5.24 – Accélérations de mm avec futures sur beignet

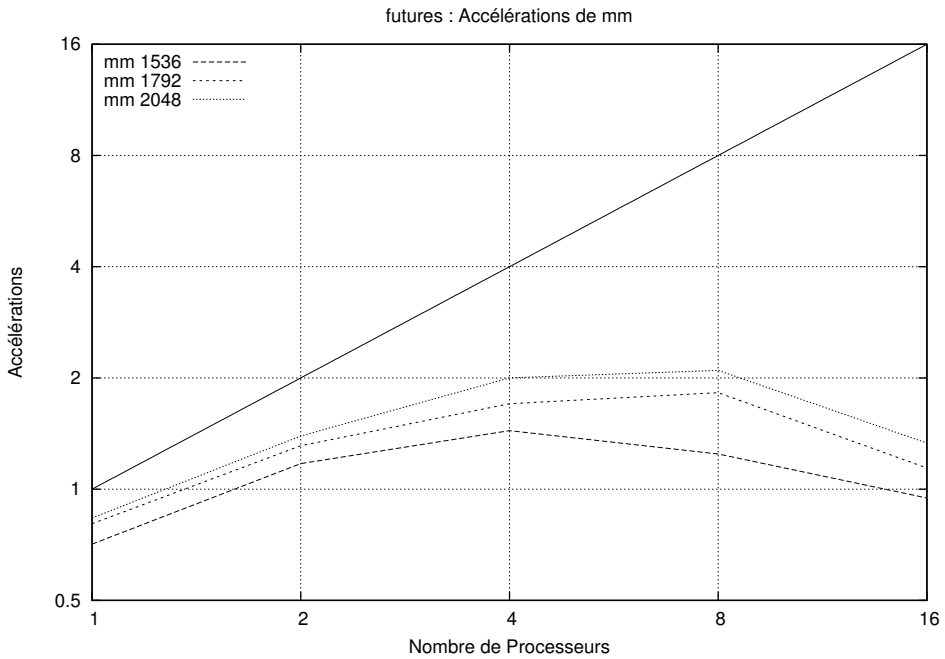

Figure 5.25 – Accélérations de mm avec futures sur clop

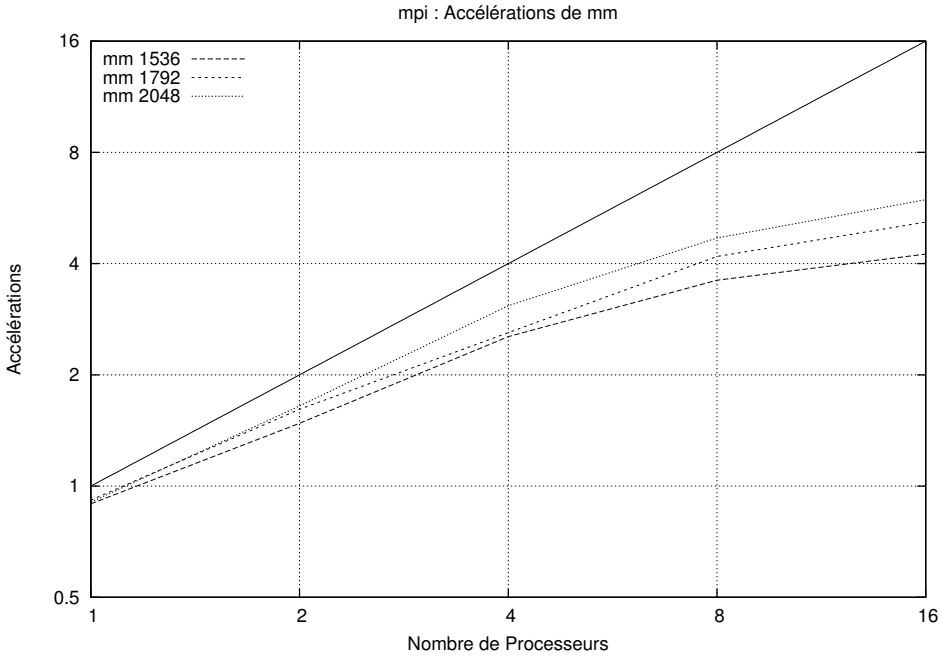

Figure 5.26 – Accélérations de mm avec MPI sur beignet

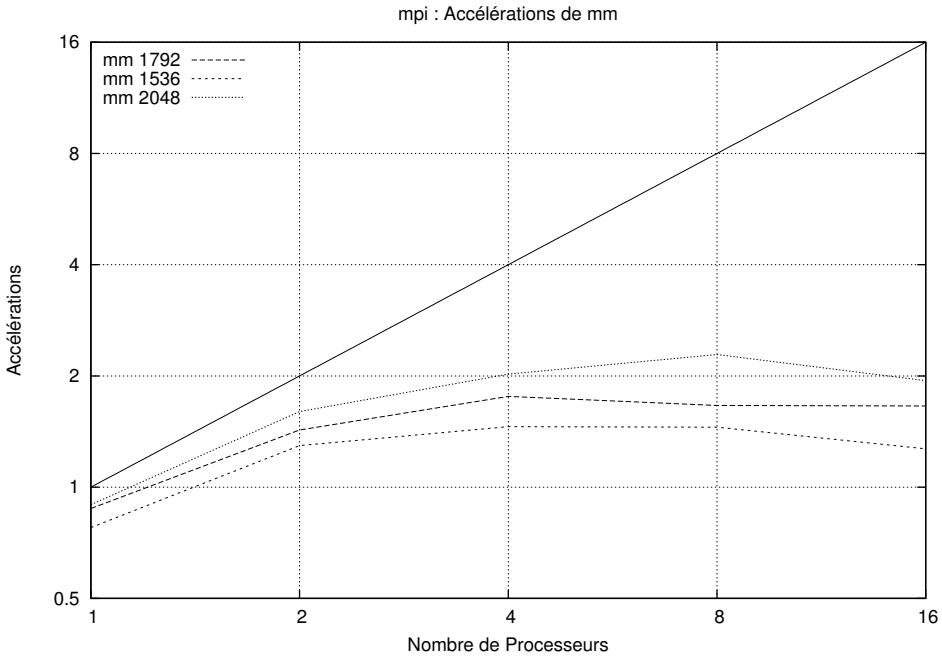

Figure 5.27 – Accélérations de mm avec MPI sur clop

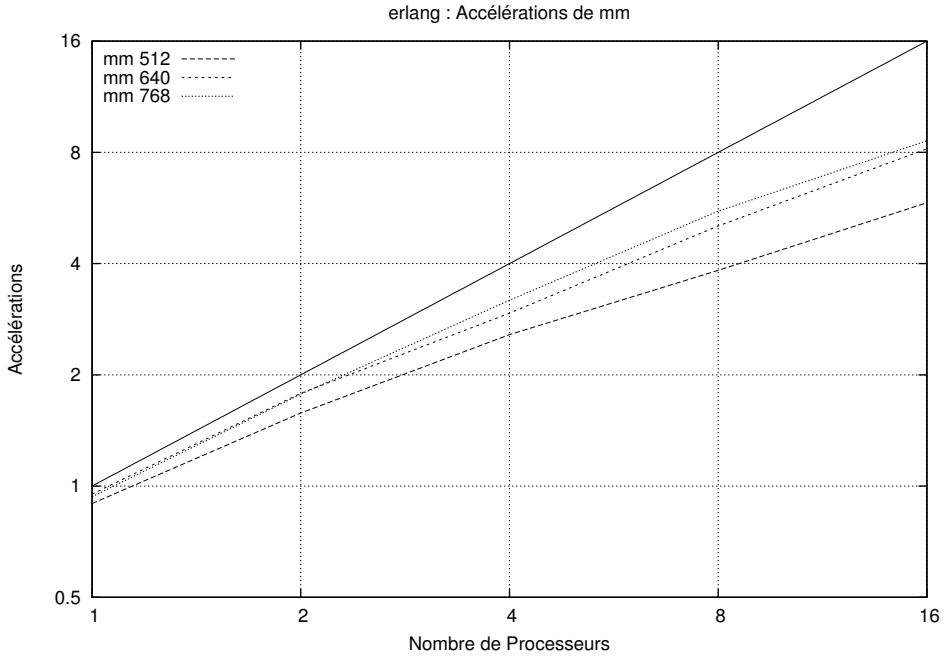

Figure 5.28 – Accélérations de mm avec Erlang sur beignet

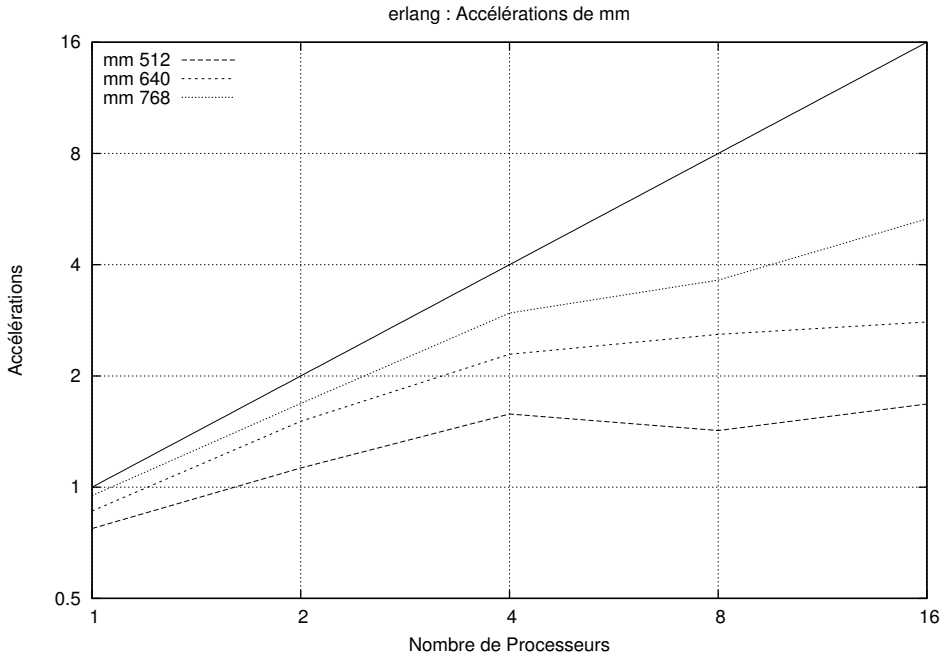

Figure 5.29 – Accélérations de mm avec Erlang sur clop

CHAPITRE 6

CONCLUSION

Le but de ce mémoire était de démontrer que l'usage d'un langage fonctionnel de haut niveau ainsi que l'implantation du partage dynamique de charge sur un langage à modèle partitionné permettaient de simplifier la conception de programmes et d'obtenir de bonnes accélérations tout en conservant des performances absolues acceptables par rapport à un langage de plus bas niveau.

Pour cela, nous avons développé un système à passage de message offrant des opérations semblables à *MPI*, mais utilisant le langage *Multilisp* sur lequel nous avons ajouté le partage dynamique de charge par vol de tâches, implanté à l'aide de la création paresseuse de tâches.

Ce mémoire apporte donc les contributions suivantes :

1. **Implantation de la création très paresseuse de tâches** : Nous implantons, dans notre système, la *création très paresseuse de tâches* qui était une idée présentée dans [13] et démontrons que cette technique peut offrir de bonnes accélérations pour des granularités élevées.

2. **Implantation d'un dialecte parallèle à modèle partitionné du langage Scheme** : Ce langage offre un sous-ensemble des opérations offertes par *MPI* et permet le développement d'algorithmes parallèles sur des systèmes à mémoire distribuée.

3. **Utilisation du partage dynamique de charge dans un langage à modèle partitionné** : Notre système permet de réduire la quantité de gestion manuelle des tâches que le programmeur doit faire dans un langage à modèle partitionné en confiant une partie de ce travail au système de balancement dynamique de charge. Il permet aussi l'utilisation de parallélisme de tâches hiérarchique.

4. **Utilisation du partage dynamique de charge par vol de tâches sur des systèmes à mémoire distribuée** : Nous démontrons qu'il est faisable d'utiliser le partage dynamique de charge par vol de tâches sur des ordinateurs à mémoire partagée et d'obtenir de bonnes accélérations et de bonnes performances.

5. **Bonnes performances relatives à MPI et Erlang** : Notre système maintient en général des performances acceptables face à *MPI* et *Erlang* tout en offrant une plus grande simplicité au niveau de la conception des programmes.

6. **Plateforme pour le développement d'un dialecte distribué de Multilisp** : En montrant que le vol de tâche est viable sur des ordinateurs à mémoire distribuée, nous ouvrons la porte au développement d'un nouveau dialecte de Multilisp offrant un modèle global de programmation ainsi que les outils nécessaires à la programmation d'algorithmes pour ces machines.

6.1 Travaux futurs

Le but ultime des travaux présentés dans ce mémoire est le développement d'un nouveau langage, inspiré de Multilisp, et offrant un modèle global comportant tous les outils et fonctionnalités nécessaires au développement d'applications pouvant s'adapter à plusieurs types d'architectures à mémoire partagée ou distribuée. Ce langage hautes-performances serait en grande partie inspiré des travaux présentés dans [19], [18], [17].

Afin de se diriger vers cet objectif, plusieurs étapes sont nécessaires. Tout d'abord, il faut refaire le système de *threads* du compilateur *Gambit-C* afin qu'il supporte le *multiplexage* des *threads* légers sur des threads systèmes. Cela permettrait d'obtenir du véritable parallélisme en mémoire partagée à l'intérieur d'une instance du programme. Pour ce faire, nous allons nous inspirer des travaux présentés dans [3].

Par la suite, nous devrons aussi retravailler le système de vol de tâches. En ce moment, il est implanté sous forme d'une librairie utilisant le langage *termite* comme plate-forme. Nous devons l'intégrer directement au système de *threads* pour le rendre

plus léger et plus performant. Il faudrait aussi trouver une meilleure manière de représenter les *placeholders* qui sont en ce moment des *promesses* cachant un *thread*

Pour supporter les fonctionnalités nécessaires à un langage parallèle à modèle global pouvant être utilisé sur des machines à mémoire distribuée, il est aussi nécessaire de supporter un *espace d'adressage global partitionné*. Nous devrons donc implanter ce concept dans *Gambit-C* afin de permettre de lier plusieurs instances de l'environnement d'exécution. Nous devrons considérer l'usage d'une librairie de communication à un sens développée pour cet usage telle que *GASNet* [6] ou *ARMCI* [22]. Cela nous permettrait aussi de réduire le coût du vol de tâche à distance.

Il faudrait aussi pouvoir effectuer du véritable parallélisme de données en implantant des structures de données distribuées et en offrant des opérations permettant d'abstraire la division des tâches. En effet, le partage dynamique de charge sans information sur la localisation des données peut occasionner d'énormes coûts de communication. Il faudrait trouver un moyen de réconcilier le parallélisme de tâche et le parallélisme de données.

Enfin, il serait intéressant d'explorer l'usage des analyses de flot de contrôle afin de pouvoir éliminer les appels à *touch* des programmes. L'analyse permettrait de détecter tous les appels à des primitives strictes risquant de recevoir un *placeholder* et ferait l'insertion des *touch* automatiquement. Nous avons déjà effectué une bonne partie du travail d'implantation de la 0-CFA [24] pour le compilateur *Gambit-C*, donc ce travail devrait suivre prochainement.

ANNEXE A

CODE DES PROGRAMMES PARALLÈLES

A.1 Multilisp

A.1.1 Fibonacci

```
(declare (standard-bindings)
         (extended-bindings)
         (block)
         (multilisp)
         (fixnum)
         (not inline)
         (not safe))

(import (user futures/futures))

(define (fib-p n)
  (if (or (= 0 n) (= 1 n))
      n
      (let ((r1 (future (fib-p (- n 1)))))
        (let ((r2 (fib-p (- n 2))))
          (+ (touch r1) r2)))))

(define-main (n)
  (let ((result (do-or-help 0 (fib-p (string->number n)))))
    (if (zero? (get-node-id))
        (println result)))
  0)
```

A.1.2 Queens

```
(declare
 (multilisp)
 (standard-bindings)
 (extended-bindings)
 (block)
 (fixnum)
 (not safe))

(import (user futures/futures))

(define (queens n)
  (let try ((rows-left n)
            (free-diag1 -1) ; all bits set
            (free-diag2 -1)
            (free-cols (- (arithmetic-shift 1 n) 1)))  ; bits 0 to n-1 set
    (let ((free (fxand free-cols free-diag1 free-diag2)))
      (let loop ((col 1))
        (cond ((> col free)
               0)
              ((= (fxand col free) 0)
               (loop (* col 2)))
              ((= rows-left 1)
               (+ 1 (loop (* col 2))))
              (else
               (let* ((sub-solns
                       (future
                        (try (- rows-left 1)
                             (+ (arithmetic-shift (- free-diag1 col) 1) 1)
                             (arithmetic-shift (- free-diag2 col) -1)
                             (- free-cols col))))
                      (other-solns (loop (* col 2))))
                 (+ (touch sub-solns) other-solns)))))))))

(define-main (n)
  (let ((n (string->number n)))
   (let ((result (do-or-help 0 (queens n))))
     (only 0 (println result)))
   0))
```

A.1.3 Mandelbrot

```
(declare
 (multilisp)
 (standard-bindings)
 (extended-bindings)
 (block)
 (not safe))

(import (user futures/futures)
        (user utils)
        (user dc))

(c-declare
#<<end-of-c

#define RADIUS2 16.0
#define MAX_COUNT 1024

end-of-c
)

(define count
  (c-lambda (double double double int int) unsigned-int8
#<<end-of-c
/* cr = r + x * step */
   const double cr = ___arg1 + ___arg4 * ___arg3;
/* ci = i + y * step */
   const double ci = ___arg2 + ___arg5 * ___arg3;

   int c;
   double zr = cr;
   double zi = ci;
   for (c = 0; c < MAX_COUNT; c++) {
     double zr2 = zr * zr;
     double zi2 = zi * zi;
     if ( zr2 + zi2 > RADIUS2 )
       break;
     else {
       zi = 2.0 * zr * zi + ci;
       zr = zr2 - zi2 + cr;
     }
   }

   ___result = c % 256;
end-of-c
))

(define (mbrot r i step n)
  (pfor 0 n (lambda (y)
```

```
              (let ((line (make-u8vector n)))
                (let loop ((x 0))
                  (if (< x n)
                      (begin
                        (u8vector-set! line x (count r i step x y))
                        (loop (+ x 1)))
                      (begin
                        (! (node 0) (vector y line))
                        #t)))))))

(define (write-matrix matrix n)
  (with-output-to-file "/tmp/futures-mandelbrot.pnm"
    (lambda ()
      (let ((intro (format "P5~%~a ~a~%255~%" n n)))
        (write-subu8vector
         (list->u8vector (map char->integer (string->list intro)))
         0
         (string-length intro))
        (write-subu8vector matrix 0 (fx* n n))))))

(define-main (n)
  (let ((n (string->number n)))
    (do-or-help 0 (mbrot -0.747 -0.16825 (/ .008192 n) n))
    (only 0 (let ((mresult (make-u8vector (* n n))))
              (let loop ((i n))
                (if (not (zero? i))
                    (recv
                     (#(row row-v)
                       (##subu8vector-move! row-v 0 n mresult (* n row))
                       (loop (- i 1))))))
              (write-matrix mresult n)))))
```

A.1.4 Multiplication de matrices

```
(declare
 (multilisp)
 (standard-bindings)
 (extended-bindings)
 (block)
 (not safe))

(import (user futures/futures)
        (user dc))

(define compute-line
  (c-lambda (int int scheme-object scheme-object scheme-object) void
#<<end-of-c
int n = ___arg1;
int line = ___arg2;

___S32 *m1 = ___CAST(___S32*,___BODY_AS(___arg3,___tSUBTYPED));
___S32 *m2 = ___CAST(___S32*,___BODY_AS(___arg4,___tSUBTYPED));
___S32 *lr = ___CAST(___S32*,___BODY_AS(___arg5,___tSUBTYPED));

___S32 *l1 = m1 + line * n;

int k,i,j;

for( k = 0; k < n; k++ )
  for( j = 0; j < n; j++ )
    lr[j] += l1[k] * m2[k*n+j];

end-of-c
))

(define m1 #f)
(define m2 #f)

(define nm1 #f)
(define nm2 #f)

(define (mm n)
  (pfor 0 n (lambda (i)
              (let ((line (make-s32vector n 0)))
                (compute-line n i m1 m2 line)
                (! (node 0) (vector i line))
                #t))))

(define-main (n)
  (let ((n (string->number n))
        (x1 1)
        (x2 1))
```

```
(set! nm1 (only 0 (make-s32vector (* n n) x1)))
(set! nm2 (only 0 (make-s32vector (* n n) x2)))
(set! m1 (broadcast nm1 0))
(set! m2 (broadcast nm2 0))
(do-or-help 0 (mm n))
(only 0 (let ((mresult (make-s32vector (* n n))))
          (let loop ((i n))
            (if (not (zero? i))
                (recv
                 (#(l line)
                  (##subs32vector-move! line 0 n mresult (* n l))
                  (loop (- i 1))))))))
0))
```

A.2 MPI

A.2.1 Fibonacci

```
#include <stdio.h>
#include <mpi.h>

typedef struct task_struct {
  int parent;
  int n;
  int poffset;
} task;

int rank;
int numprocs;

MPI_Datatype Task_type;

long fib_seq( int n ) {
  if (n == 0 || n == 1)
    return n;
  else
    return fib_seq(n-1) + fib_seq(n-2);
}

long fib_par(int n, int poffset) {

  int next = rank + poffset;

  if( next < numprocs ) {
    long r1;
    long r2;
    task t = {rank,n-1,poffset << 1};

    MPI_Send(&t, 1, Task_type, next, 0, MPI_COMM_WORLD);
    r1 = fib_par(n-2, poffset << 1);
    MPI_Recv(&r2, 1, MPI_LONG, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IG

    return r1 + r2;
  }
  else
    return fib_seq(n);

}

void prepare_types() {
  MPI_Aint a1;
  MPI_Aint a2;

  task t;
```

```
  MPI_Datatype task_types[1];
  int task_blocks[1];
  MPI_Aint task_disps[1];

  task_types[0] = MPI_INT;
  task_blocks[0] = 3;

  MPI_Get_address(&t.parent,&a1);
  MPI_Get_address(&t,&a2);
  task_disps[0] = a1 - a2;

  MPI_Type_create_struct(1,task_blocks,task_disps,task_types,&Task_type);
  MPI_Type_commit(&Task_type);
}

int main(int argc, char *argv[]) {

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  prepare_types();

  numprocs = atoi(argv[1]);

  if( rank == 0 ) {
    int n =  atoi(argv[2]);
    printf("%ld\n", fib_par(n,1));
  }
  else if (rank < numprocs) {
    int r;
    task t;
    MPI_Recv(&t, 1, Task_type, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    r = fib_par(t.n,t.poffset);
    MPI_Send(&r, 1, MPI_LONG, t.parent, 0, MPI_COMM_WORLD);
  }

  MPI_Finalize();
}
```

A.2.2 Queens

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct task_struct {
  int parent;
  int col;
  int cols;
  int rows_left;
  int poffset;
  int free;
  int free_diag1;
  int free_diag2;
  int free_cols;
} task;

int rank;
int numprocs;
int n;

MPI_Datatype Task_type;

int queens_seq(int col, int rows_left, int free, int free_diag1, int free_diag2, int free
  int res = 0;

  if ( rows_left > 0 ) {
    while (col <= free) {
      if ( col & free ) {
        int nfree_diag1 = ((free_diag1 - col) << 1) + 1;
        int nfree_diag2 = (free_diag2 - col) >> 1;
        int nfree_cols = free_cols-col;
        int nfree = nfree_diag1 & nfree_diag2 & nfree_cols;
        res += queens_seq(1,rows_left-1,nfree,nfree_diag1,nfree_diag2,nfree_cols);
      }
      col = col << 1;
    }
  }
  else {
    while (col <= free) {
      if (col & free)
        res++;
      col = col << 1;
    }
  }

  return res;

}
```

```
int queens_par( int col, int cols, int rows_left, int poffset, int free, int free_diag1, int free_d

  int next = rank + poffset;
  int shift = cols/2;

  if (next >= numprocs)
    return queens_seq(col,rows_left,free,free_diag1,free_diag2,free_cols);
  else if (!shift) {
    int nfree_diag1 = ((free_diag1 - col) << 1) + 1;
    int nfree_diag2 = (free_diag2 - col) >> 1;
    int nfree_cols = free_cols-col;
    int nfree = nfree_diag1 & nfree_diag2 & nfree_cols;
    return queens_par(1,n,rows_left-1,poffset,nfree,nfree_diag1,nfree_diag2,nfree_cols);
  }
  else {

    int r1;
    int r2;
    int ncol = col << shift;
    int ncols = cols - shift;
    task t = {rank,ncol,ncols,rows_left,poffset << 1,free,free_diag1,free_diag2,free_cols};

    MPI_Send(&t,1,Task_type,next,0,MPI_COMM_WORLD);

    r1 = queens_par(col,shift,rows_left,poffset << 1,free & (ncol-1),free_diag1,free_diag2,free_col

    MPI_Recv(&r2,1,MPI_INT,next,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

    return r1+r2;

  }

}

void prepare_types() {
  MPI_Aint a1;
  MPI_Aint a2;

  task t;
  MPI_Datatype task_types[1];
  int task_blocks[1];
  MPI_Aint task_disps[1];

  task_types[0] = MPI_INT;
  task_blocks[0] = 9;

  MPI_Get_address(&t.parent,&a1);
  MPI_Get_address(&t,&a2);
  task_disps[0] = a1 - a2;
```

```
  MPI_Type_create_struct(1,task_blocks,task_disps,task_types,&Task_type);
  MPI_Type_commit(&Task_type);
}

int main(int argc, char *argv[]) {

  int parallel;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  numprocs = atoi(argv[1]);
  n = atoi(argv[2]);

  prepare_types();

  if( rank == 0 ) {
    int free_cols = (1 << n) - 1;
    int result = queens_par(1,n,n-1,1,free_cols,-1,-1,free_cols);
    printf("%d\n", result);
  }
  else if (rank < numprocs) {
    int result;
    task t;
    MPI_Recv(&t, 1, Task_type, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IG
    result = queens_par(t.col,t.cols,t.rows_left,t.poffset,t.free,t.free_diag1,t.free_dia
    MPI_Send(&result, 1, MPI_INT, t.parent, 0, MPI_COMM_WORLD);
  }

  MPI_Finalize();

}
```

A.2.3 Mandelbrot

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#include <stdlib.h>
#include <stdint.h>

#define MAX_COUNT 1024
#define RADIUS2 16.0

int rank;
int numprocs;

MPI_Comm communicator;

void write_result( uint8_t *result, int n ) {
  FILE *fp = fopen("/tmp/mpi-mandelbrot.pnm", "w+");
  fprintf(fp, "P5\n%d %d\n255\n", n, n);
  fwrite(result, sizeof(uint8_t), n*n, fp);
  fclose(fp);
}

uint8_t count( double r, double i, double step, int x, int y ) {

  double cr = r + x * step;
  double ci = i + y * step;

  int c;
  double zr = cr;
  double zi = ci;

  for ( c = 0; c < MAX_COUNT; c++ ) {
    double zr2 = zr * zr;
    double zi2 = zi * zi;
    if ( zr2 + zi2 > RADIUS2 )
      break;
    zi = 2.0 * zr * zi + ci;
    zr = zr2 - zi2 + cr;
  }

  return c % 256;

}

void mandelbrot(double r, double i, double step, int n) {

  uint8_t *result;
  uint8_t *result_ptr;
```

```
  int numlines = n / numprocs;
  uint8_t *lines = malloc(n * numlines * sizeof(uint8_t));
  uint8_t *current_line = lines;
  int x,y;

  for( y = rank; y < n; y += numprocs ) {

    for( x = 0; x < n; x++ )
      *(current_line+x) = count(r,i,step,x,y);

    current_line += n;

  }

  current_line = lines;

  if( rank == 0 ) {
    result = malloc(n * n * sizeof(uint8_t));
    result_ptr = result;
  }

  for( x = 0; x < numlines; x++ ) {

    MPI_Gather( current_line, n, MPI_BYTE, result, n, MPI_BYTE, 0, communicator );

    current_line += n;

    if( rank == 0 )
      result += n * numprocs;

  }

  if( rank == 0 )
    write_result( result_ptr, n );

}

void make_communicator() {

  int i;
  MPI_Group original;
  MPI_Group new;
  int *ranks = (int *)malloc(numprocs*sizeof(int));

  for( i = 0; i < numprocs; i++ )
    ranks[i] = i;

  MPI_Comm_group(MPI_COMM_WORLD,&original);

  MPI_Group_incl(original,numprocs,ranks,&new);
```

```
  MPI_Comm_create(MPI_COMM_WORLD,new,&communicator);

}

int main(int argc, char *argv[]) {

  int n;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  numprocs = atoi(argv[1]);
  n = atoi(argv[2]);

  make_communicator();

  if (rank < numprocs)
    mandelbrot( -0.747, -0.16825, .008192 / n, n);

  MPI_Finalize();
}
```

A.2.4 Multiplication de matrices

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#include <stdint.h>
#include <stdlib.h>

int rank;
int numprocs;
MPI_Comm communicator;

int *m1;
int *m2;
int *mr;

int numlines;

void distribute_work(int n, int *nm1) {

  MPI_Bcast( m2, n*n, MPI_INT, 0, communicator );

  MPI_Scatter( nm1, n*numlines, MPI_INT, m1, n*numlines, MPI_INT, 0, communicator );

}

void gather_work( int n, int *nmr ) {

  MPI_Gather( mr, n*numlines, MPI_INT, nmr, n*numlines, MPI_INT, 0, communicator );

}

void multiply( int n, int *m1, int *m2, int *mr ) {

  int k,i,j;

  for( i = 0; i < numlines; i++ )
    for( j = 0; j < n; j++ )
      mr[i*n+j] = 0;

  for( i = 0; i < numlines; i++ )
    for( k = 0; k < n; k++ )
      for( j = 0; j < n; j++ )
        mr[i*n+j] += m1[i*n+k] * m2[k*n+j];

}

void make_communicator() {

  int i;
```

```
  MPI_Group original;
  MPI_Group new;
  int *ranks = (int *)malloc(numprocs*sizeof(int));

  for( i = 0; i < numprocs; i++ )
    ranks[i] = i;

  MPI_Comm_group(MPI_COMM_WORLD,&original);

  MPI_Group_incl(original,numprocs,ranks,&new);

  MPI_Comm_create(MPI_COMM_WORLD,new,&communicator);

}

int main(int argc, char *argv[]) {

  int n;
  int x1 = 1;
  int x2 = 1;

  int *nm1;
  int *nmr;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  numprocs = atoi(argv[1]);
  n = atoi(argv[2]);

  make_communicator();

  if( rank < numprocs ) {

    numlines = n / numprocs;

    m1 = malloc(numlines * n * sizeof(int));
    m2 = malloc(n * n * sizeof(int));
    mr = malloc(numlines * n * sizeof(int));

    if( rank == 0 ) {
      int i;

      nm1 = malloc( n * n * sizeof(int) );
      nmr = malloc( n * n * sizeof(int) );

      for( i = 0; i < n*n; i++ ) {
        nm1[i] = x1;
        m2[i] = x2;
      }
```

```
    }

    distribute_work(n, nm1);

    multiply(n,m1,m2,mr);

    gather_work(n, nmr);

  }

  MPI_Finalize();
}
```

A.3 Erlang

A.3.1 Fibonacci

```
-module(fib).
-export([fib/1,fibs/4,par/1,seq/1]).

fib(0) -> 0;
fib(1) -> 1;
fib(N) ->
    fib(N-1) + fib(N-2).

fibs_receive(Parent) ->
    receive
        N1 ->
            receive
                N2 ->
                    Parent ! N1 + N2
            end
    end.

fibs(0,Parent,_,_) -> Parent ! 0;
fibs(1,Parent,_,_) -> Parent ! 1;
fibs(N,Parent,1,_) ->
    Parent ! fib(N);
fibs(N,Parent,Procs,[]) ->
    spawn_link(fib,fibs,[N-1,self(),Procs div 2,[]]),
    spawn_link(fib,fibs,[N-2,self(),Procs div 2,[]]),
    fibs_receive(Parent);
fibs(N,Parent,Procs,[Node|Nodes]) ->
    {Nodes1,Nodes2} = lists:split(length(Nodes) div 2,Nodes),
    spawn_link(fib,fibs,[N-1,self(),Procs div 2,Nodes1]),
    spawn_link(Node,fib,fibs,[N-2,self(),Procs div 2,Nodes2]),
    fibs_receive(Parent).

par(ARGS) ->
    {Ns,Procs,[N]} = cluster:startup(ARGS),
    case Ns of
        [] ->
            spawn_link(fib,fibs,[N,self(),Procs,[]]);
        [Node|Nodes] ->
            spawn_link(Node,fib,fibs,[N,self(),Procs,Nodes])
    end,
    receive
        R when is_integer(R) ->
            io:format("~w~n",[R])
    end,
    ok = cluster:halt_nodes(Ns).
```

```
seq([Nstr]) ->
    {N,[]} = string:to_integer(Nstr),
    io:format("~w~n",[fib(N)]).
```

A.3.2 Queens

```
-module(queens).
-export([par/1,seq/1,queens_par/9]).
-define(Make_diag1(Col,Diag1), Diag1 - Col bsl 1 + 1).
-define(Make_diag2(Col,Diag2), Diag2 - Col bsr 1).
-define(Make_cols(Col,Cols), Cols - Col).
-define(Make_free(Diag1,Diag2,Cols), Diag1 band Diag2 band Cols).

queens_seq(Col,_,Free,_,_,_) when Col > Free ->
    0;
queens_seq(Col,Rows_left,Free,Diag1,Diag2,Cols) when Col band Free == 0 ->
    queens_seq(Col bsl 1,Rows_left,Free,Diag1,Diag2,Cols);
queens_seq(Col,Rows_left,Free,Diag1,Diag2,Cols) when Rows_left == 0 ->
    1 + queens_seq(Col bsl 1,Rows_left,Free,Diag1,Diag2,Cols);
queens_seq(Col,Rows_left,Free,Diag1,Diag2,Cols) ->
    NDiag1 = ?Make_diag1(Col,Diag1),
    NDiag2 = ?Make_diag2(Col,Diag2),
    NCols  = ?Make_cols(Col,Cols),
    NFree  = ?Make_free(NDiag1,NDiag2,NCols),
    Subs = queens_seq(1,Rows_left-1,NFree,NDiag1,NDiag2,NCols),
    Subs + queens_seq(Col bsl 1,Rows_left,Free,Diag1,Diag2,Cols).

queens_receive(Parent) ->
    receive
        R1 when is_integer(R1) ->
            receive
                R2 when is_integer(R2) ->
                    Parent ! R1 + R2
            end
    end.

queens_par(Col,_,Rows_left,_,Free,Diag1,Diag2,Cols,{Parent,1,_}) ->
    Parent ! queens_seq(Col,Rows_left,Free,Diag1,Diag2,Cols);
queens_par(Col,1,Rows_left,N,_,Diag1,Diag2,Cols,Par_settings) ->
    NDiag1 = ?Make_diag1(Col,Diag1),
    NDiag2 = ?Make_diag2(Col,Diag2),
    NCols  = ?Make_cols(Col,Cols),
    NFree  = ?Make_free(NDiag1,NDiag2,NCols),
    queens_par(1,N,Rows_left-1,N,NFree,NDiag1,NDiag2,NCols,Par_settings);
queens_par(Col,Cols_left,Rows_left,N,Free,Diag1,Diag2,Cols,{Parent,Procs,Nodes}) ->
    Shift = Cols_left div 2,
    NProcs = Procs div 2,
    NCol = Col bsl Shift,
    NCols_left = Cols_left - Shift,
    NFree = Free band (NCol - 1),
    case Nodes of
        [] -> NSettings = {self(),NProcs,[]},
              spawn_link(queens,queens_par,[Col,Shift,Rows_left,N,NFree,Diag1,Diag2,Cols,NSettings]
              spawn_link(queens,queens_par,[NCol,NCols_left,Rows_left,N,Free,Diag1,Diag2,Cols,NSett
```

```
        [Node|Rest] -> {Nodes1,Nodes2} = lists:split(length(Rest) div 2,Rest),
                       spawn_link(queens,queens_par,[Col,Shift,Rows_left,N,NFree,Diag1,Di
                       spawn_link(Node,queens,queens_par,[NCol,NCols_left,Rows_left,N,Fre
    end,
    queens_receive(Parent).

par(ARGS) ->
    {Nodes,Procs,[N]} = cluster:startup(ARGS),
    Cols = 1 bsl N - 1,
    case Nodes of
        [] ->
            spawn_link(queens,queens_par,[1,N,N-1,N,Cols,-1,-1,Cols,{self(),Procs,[]}]);
        [Node|Rest] ->
            spawn_link(Node,queens,queens_par,[1,N,N-1,N,Cols,-1,-1,Cols,{self(),Procs,Re
    end,
    receive
        R when is_integer(R) ->
            io:format("~w~n",[R])
    end,
    ok = cluster:halt_nodes(Nodes).

seq([Nstr]) ->
    {N,[]} = string:to_integer(Nstr),
    Cols = 1 bsl N - 1,
    io:format("~w~n",[queens_seq(1,N-1,Cols,-1,-1,Cols)]).
```

A.3.3 Mandelbrot

```
-module(mandelbrot).
-export([par/1,seq/1,mbrot_par/7]).

-define(Max_count,1024).
-define(Radius2,16.0).
-define(R,-0.747).
-define(I,-0.16825).
-define(Basestep,0.008192).

count_loop(C,_,_,_,_) when C >= ?Max_count ->
    C;
count_loop(C,Cr,Ci,Zr,Zi) ->
    Zr2 = Zr * Zr,
    Zi2 = Zi * Zi,
    if Zr2 + Zi2 > ?Radius2 ->
            C;
       true ->
            New_Zr = Zr2 - Zi2 + Cr,
            New_Zi = 2.0 * Zr * Zi + Ci,
            count_loop(C+1,Cr,Ci,New_Zr,New_Zi)
    end.

count(R,I,Step,X,Y) ->
    Cr = R + X * Step,
    Ci = I + Y * Step,
    count_loop(0,Cr,Ci,Cr,Ci) rem 256.

mbrot_line(_,-1,_,Line) ->
    Line;
mbrot_line({R,I,Step,_} = Params,X,Y,Line) ->
    Result = count(R,I,Step,X,Y),
    mbrot_line(Params,X-1,Y,[Result|Line]).

mbrot_par(_,Parent,_,_,_,Y,Lines) when Y < 0 ->
    Parent ! Lines;
mbrot_par({_,_,_,N} = Params,Parent,Procs,Delta,Offset,Y,Lines) ->
    Line = {Offset,mbrot_line(Params,N-1,Y,[])},
    mbrot_par(Params,Parent,Procs,Delta,Offset-Delta,Y-Procs,[Line|Lines]).

mbrot_seq(_,Y,Lines) when Y < 0 ->
    Lines;
mbrot_seq({_,_,_,N} = Params,Y,Lines) ->
    Line = mbrot_line(Params,N-1,Y,[]),
    mbrot_seq(Params,Y-1,[Line|Lines]).

make_params(N) ->
    {?R,?I,?Basestep/N,N}.
```

```
spawn_pars([],_,_,_,_,_,_) ->
    ok;
spawn_pars([Node|Nodes],{_,_,_,N} = Params,Parent,Procs,Delta,Offset,Y) ->
    spawn_link(Node,mandelbrot,mbrot_par,[Params,Parent,Procs,Delta,Offset,Y,[]]),
    spawn_pars(Nodes,Params,Parent,Procs,Delta,Offset-N,Y-1).

par_receive(0,LocBytes) ->
    LocBytes;
par_receive(Procs,LocBytes) ->
    receive
        Lines when is_list(Lines) ->
            par_receive(Procs-1,lists:append(Lines,LocBytes))
    end.
par_receive(Procs) -> par_receive(Procs,[]).

par(ARGS) ->
    {Ns,P,[N]} = cluster:startup(ARGS),
    Nodes = cluster:explicit_nodes(Ns,P),
    Header = io_lib:format("P5~n~w ~w~n255~n",[N,N]),
    Offset = N*(N-1)+length(Header),
    spawn_pars(Nodes,make_params(N),self(),P,P*N,Offset,N-1),
    {ok,File} = file:open("/tmp/erlang-mandelbrot.pnm",[write,raw,delayed_write]),
    ok = file:write(File,Header),
    ok = file:pwrite(File,par_receive(length(Nodes))),
    ok = file:close(File),
    ok = cluster:halt_nodes(Ns).

seq([Nstr]) ->
    {N,[]} = string:to_integer(Nstr),
    Result = mbrot_seq(make_params(N),N-1,[]),
    {ok,File} = file:open("/tmp/erlang-mandelbrot.pnm",[write,raw,read_ahead]),
    ok = file:write(File,io_lib:format("P5~n~w ~w~n255~n",[N,N])),
    ok = file:write(File,Result),
    ok = file:close(File).
```

A.3.4 Multiplication de matrices

```
%% Inspired from http://dada.perl.it/shootout/matrix_allsrc.html

-module(mm).
-export([seq/1,par/1,par_mmult/4]).

sumprod(0, _, _, Sum, _, _) -> Sum;
sumprod(I, C, R, Sum, M1, M2) ->
    NewSum = Sum + (element(I,element(R,M1)) * element(C,element(I,M2))),
    sumprod(I-1, C, R, NewSum, M1, M2).

rowmult(_, 0, _, L, _, _) -> list_to_tuple(L);
rowmult(I, C, R, L, M1, M2) ->
    SumProd = sumprod(I, C, R, 0, M1, M2),
    rowmult(I, C-1, R, [SumProd|L], M1, M2).

mmult(_, _, 0, MM, _, _) -> list_to_tuple(MM);
mmult(I, C, R, MM, M1, M2) ->
    NewRow = rowmult(I, C, R, [], M1, M2),
    mmult(I, C, R-1, [NewRow|MM], M1, M2).

mmult(M1, M2) ->
    Inner = size(M2),
    NRows = size(M1),
    NCols = size(element(1,M2)),
    mmult(Inner, NCols, NRows,[], M1, M2).

par_mmult(Parent,SM1,M2,Offset) ->
    M1 = list_to_tuple(SM1),
    MR = dematrix(mmult(M1,M2)),
    Parent ! {Offset,MR}.

mkrow(0, L, _) -> list_to_tuple(lists:reverse(L));
mkrow(N, L, Count) -> mkrow(N-1, [Count|L], Count).

mkmatrix(0, _, _, M) -> lists:reverse(M);
mkmatrix(NR, NC, Count, M) ->
    Row = mkrow(NC, [], Count),
    mkmatrix(NR-1, NC, Count, [Row|M]).

mkmatrix(NR, NC) -> list_to_tuple(mkmatrix(NR, NC, 1, [])).

mklistmatrix(NR,NC) -> mkmatrix(NR,NC,1,[]).

dematrix(M) -> lists:map(fun tuple_to_list/1,tuple_to_list(M)).

mksubmatrix(M,0) ->
    {[],M};
mksubmatrix([L1|R],N) ->
```

```
    {Sub,Rest} = mksubmatrix(R,N-1),
    {[L1|Sub],Rest}.

mksubmatrices([],_) -> [];
mksubmatrices(M,Size) ->
    {Sub,Rest} = mksubmatrix(M,Size),
    [Sub|mksubmatrices(Rest,Size)].

spawn_pars(_,[],[],_,[]) ->
    ok;
spawn_pars(Parent,[Node|Nodes],[SM1|M1],M2,[Offset|Offsets]) ->
    spawn_link(Node,mm,par_mmult,[Parent,SM1,M2,Offset]),
    spawn_pars(Parent,Nodes,M1,M2,Offsets).

integer_to_32bits(I) ->
    [(I bsr 24) band 255,(I bsr 16) band 255,(I bsr 8) band 255,I band 255].

integers_to_bytes(L) ->
    lists:map(fun integer_to_32bits/1,lists:flatten(L)).

par_receive(0,LocBytes) ->
    LocBytes;
par_receive(Procs,LocBytes) ->
    receive
        {N,L} when is_integer(N), is_list(L) ->
            Bytes = integers_to_bytes(L),
            par_receive(Procs-1,[{N,Bytes}|LocBytes])
    end.
par_receive(Procs) -> par_receive(Procs,[]).

par(ARGS) ->
    {Ns,P,[N]} = cluster:startup(ARGS),
    Nodes = cluster:explicit_nodes(Ns,P),
    Size = N div length(Nodes),
    M1 = mksubmatrices(mklistmatrix(N,N),Size),
    M2 = mkmatrix(N,N),
    Offsets = lists:seq(0,N*N*4-1,N*4*Size),
    spawn_pars(self(),Nodes,M1,M2,Offsets),
    Result = par_receive(length(Nodes)),
    ok = cluster:halt_nodes(Ns),
    Result.

seq([Nstr]) ->
    {N,[]} = string:to_integer(Nstr),
    M1 = mkmatrix(N,N),
    M2 = mkmatrix(N,N),
    Result = integers_to_bytes(dematrix(mmult(M1,M2))),
    {ok,File} = file:open("/tmp/erlang-mm-seq",[write,raw,delayed_write]),
    ok = file:write(File,Result),
    ok = file:close(File).
```

ANNEXE B

PROFILS D'EXÉCUTION DES PROGRAMMES

Cette annexe contient les profils d'exécution des noeuds lors d'une exécution de chaque tests en utilisant 16 processeurs sur Beignet. Cinq états différents sont représentés dans ces profils :

1. **Work** : Lorsqu'un processeur effectue du travail utile, c'est-à-dire qu'il se trouve dans le thread travailleur.

2. **Thread-manager** : Lorsque le processeur se trouve dans le thread manager. Il peut être en train de voler une tâche, répondre à une demande de vol, ou bien recevoir un message de son travailleur.

3. **Placeholder** : Lorsqu'un placeholder est en train de répondre à une requête sur sa valeur où bien se fait déterminer.

4. **Unknown** : L'ensemble des autres threads exécutée dans le système. Cela inclut principalement les proxy du système termite qui se charge de transmettre les messages au bon processus et de sérialiser. Le gros du travail dans cet état est la sérialisation des données.

5. **Idle** : Lorsqu'aucun thread n'est exécuté par le système. Il n'y a alors pas de travailleurs et le thread-manager est en attente soit sur un vol ou bien tout simplement inactif.

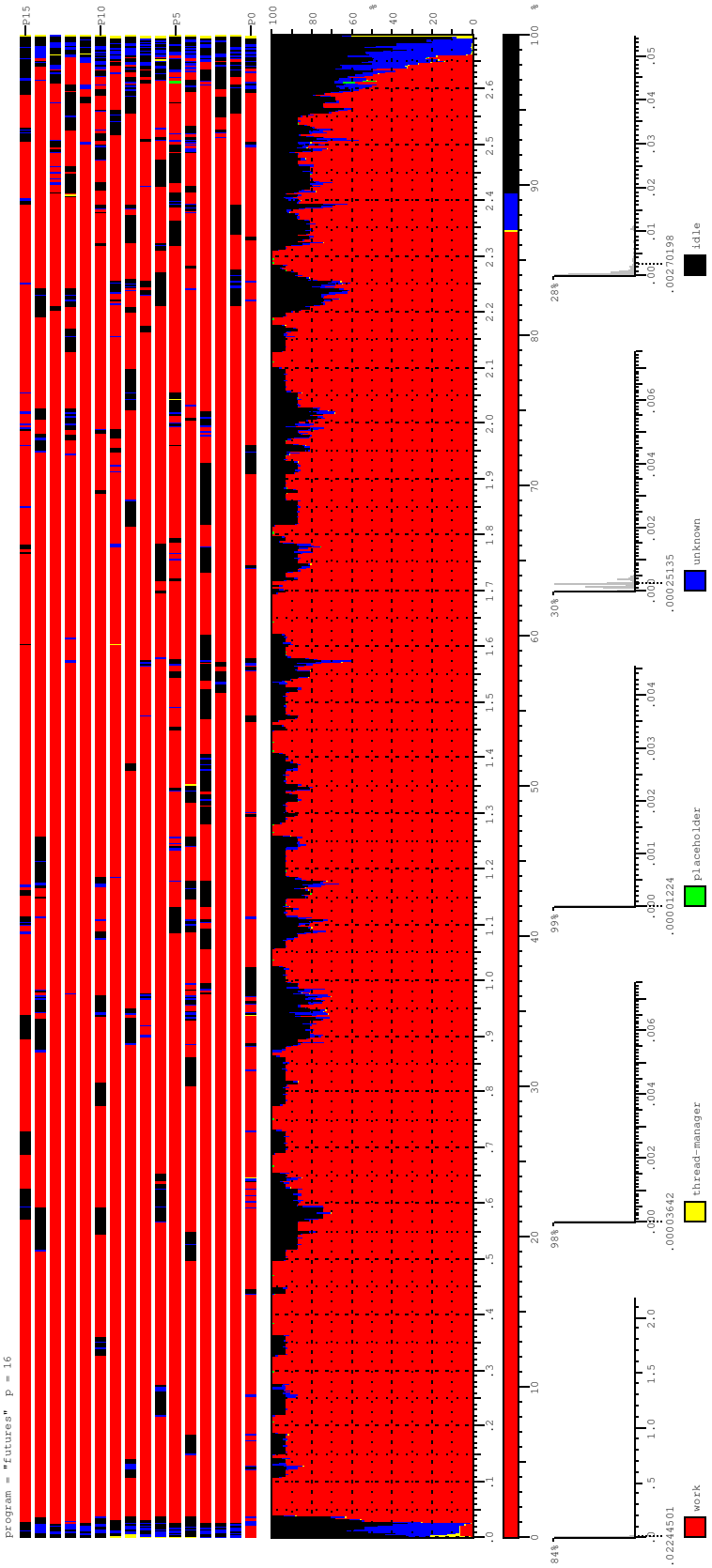

Figure B.1 – Profile d'exécution de fibonacci 44

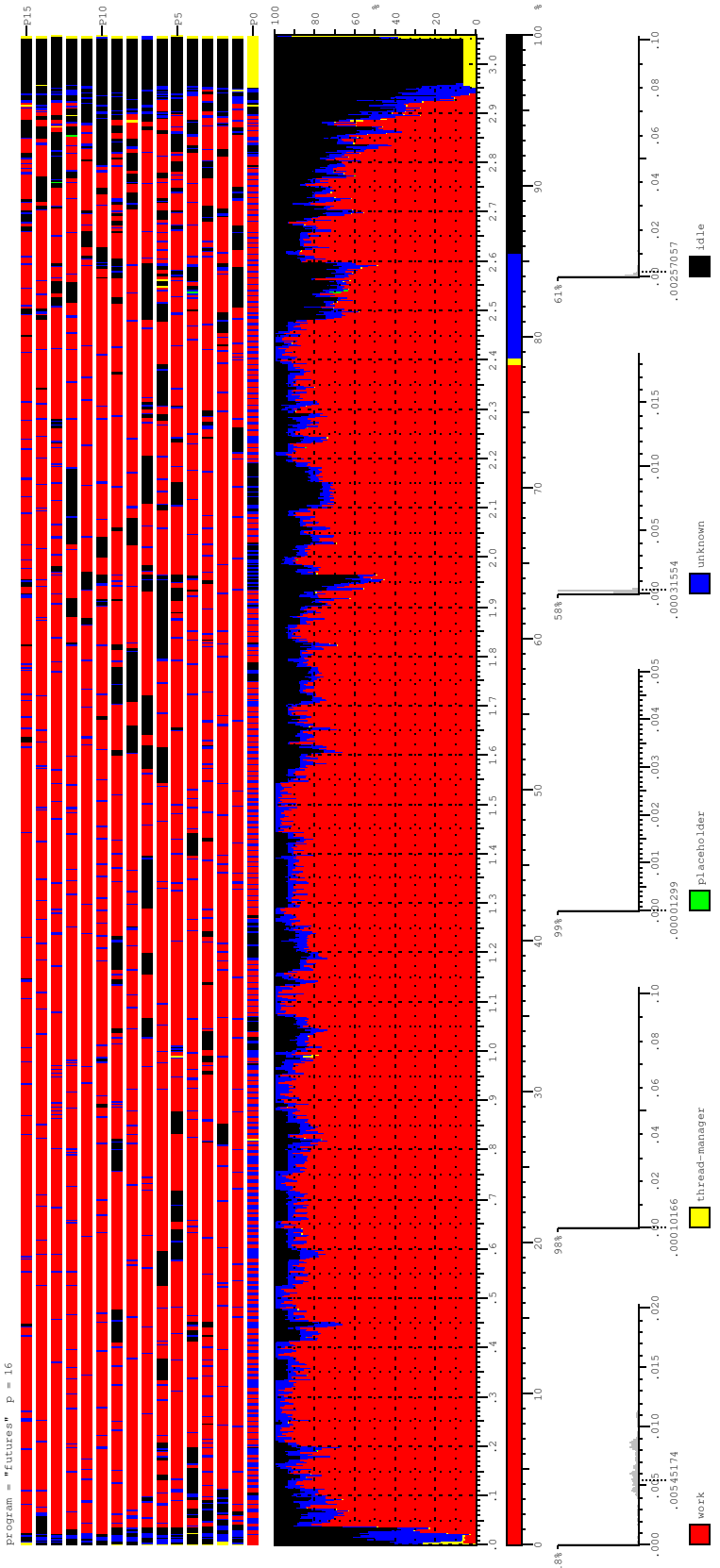

Figure B.2 – Profile d'exécution de mandelbrot 5120

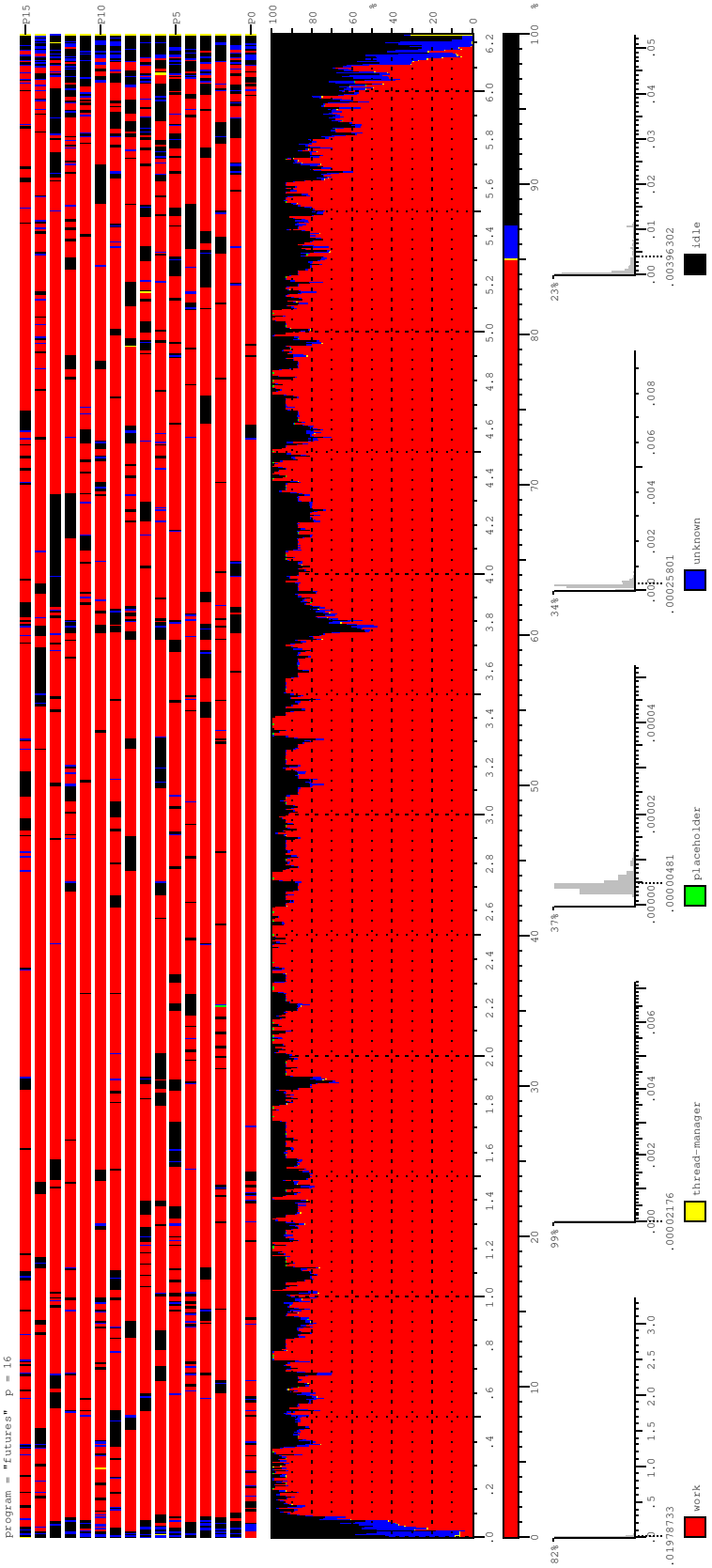

Figure B.3 – Profile d'exécution du démarrage de queens 16

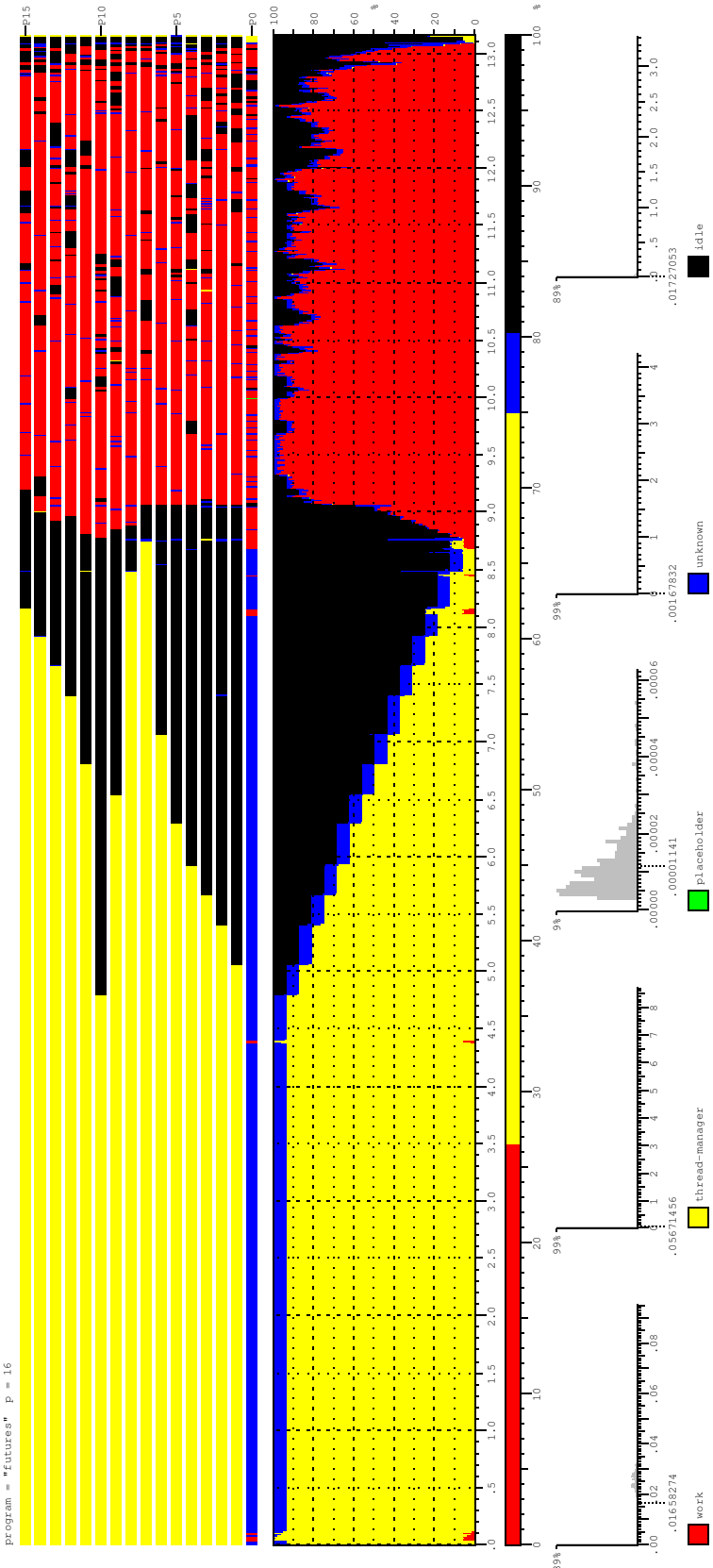

Figure B.4 – Profile d'exécution de mm 2048

BIBLIOGRAPHIE

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams Iv, D. P. Friedman, E. Kohlbecker, G. L. Steele, Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman et M. Wand. Revised report on the algorithmic language scheme. *Higher Order Symbol. Comput.*, 11(1):7–105, 1998. ISSN 1388-3690.

[2] Gigabit Ethernet Alliance. `http://www.gigabit-ethernet.org/`.

[3] Nimar S. Arora, Robert D. Blumofe et C. Greg Plaxton. Thread scheduling for multiprogrammed muliprocessors. *Theory of Computer Systems*, 34(2):115–144, 2001.

[4] Infiniband Trade Association. `http://www.infinibandta.org/`.

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

[6] Dan Bonachea. Gasnet specification v1.1. Rapport technique, Octobre 2002.

[7] David Callahan, Bradford L. Chamberlain et Hans P. Zima. The cascade high productivity language. Dans *International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60. IEEE Computer Society, Avril 2004.

[8] B.L. Chamberlain, D. Callahan et H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, Août 2007.

[9] William D. Clinger et Eric M. Hartheimer, Anne H. et Ost. Implementation strategies for first-class continuations. *Higher Order Symbol. Comput.*, 12(1):7–45, 1999. ISSN 1388-3690.

[10] Kemal Ebcioglu, Vijay Saraswat et Vivek Sarkar. X10 : An experimental language for high productivity programming of scalable systems. Dans *P-PHEC workshop*. HPCA, 2005.

[11] T. A. El-Ghazawi, W. W. Carlson et J. M. Draper. Upc language specification v1.1.1, Octobre 2003.

[12] Marc Feeley. Gambit-c version 4. http ://www.iro.umontreal.ca/ gambit/.

[13] Marc Feeley. *An efficient and general implementation of futures on large scale shared-memory multiprocessors*. Thèse de doctorat, Brandeis University, 1993.

[14] Guillaume Germain. Concurrency oriented programming in termite scheme. Dans *ERLANG '06 : Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 20–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1.

[15] Robert H. Halstead, Jr. Multilisp : a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985. ISSN 0164-0925.

[16] Maurice Herlihy, Victor Luchangco et Mark Moir. A flexible framework for implementing software transactional memory. Dans *CM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Octobre 2006.

[17] Cray Inc. Chapel language specification 0.796, Octobre 2010.

[18] IBM Inc. Report on the programming language x10, version 2.1, Octobre 2010.

[19] Sun Microsystems Inc. The fortress language specification, version 1.0 beta, March 2007.

[20] Eric Mohr. *Dynamic partitioning of parallel Lisp programs*. Thèse de doctorat, Yale University, 1992.

[21] Forum MPI. Mpi : A message-passing interface. Rapport technique, 1994.

[22] Jarek Nieplocha et Bryan Carpenter. Armci : A portable remote memory copy library for distributed array libraries and compiler run-time systems. Dans *Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming*, pages 533–546, Avril 1999.

[23] R. W. Numrich et J. K. Reid. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, Août 1998.

[24] Olin Shivers. Control flow analysis in scheme. Dans *ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 164–174, Juin 1988.

[25] Guy Steele. Parallel programming and parallel abstractions in fortress. Dans *Proceedings of the fourteenth conference of Parallel Architectures and Compilation*. PACT, Septembre 2005.

[26] Robert Virding, Claes Wikström et Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996. ISBN 0-13-508301-X.

[27] Michèle Weiland. Chapel, fortress and x10 : novel languages for hpc. Rapport technique, Octobre 2007.

[28] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamato, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella et Alex Aiken. Titanium : A high-performance java dialect. *Concurrency : Practice and Experience*, 10(11-13):825–836, Septembre 1998.