INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THESE

pour obtenir le titre de

DOCTEUR DE L'INPG

Spécialité : "INFORMATIQUE : SYSTEMES ET COMMUNICATIONS"

présentée et soutenue publiquement

par

Alexandre da Silva CARISSIMI

le 25 Novembre 1999

ATHAPASCAN-0 : Exploitation de la multiprogrammation légère sur grappes de multiprocesseurs

---

Directeur de thèse :

M. Jacques BRIAT
Mme Brigitte PLATEAU

Composition du jury :

Examinateurs : M. Roland BALTER, *Président*
M. Jacques BRIAT
M. Jean-François MEHAUT
Mme Brigitte PLATEAU
M. Xavier ROUSSET DE PINA

Rapporteurs : M. Jean-Marc GEIB
M. Jean-Louis PAZAT

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

À mes parents,
Lélio et Nelcy

Remerciements

Voici ce que j'ai appris pendant mon doctorat : « La route est plus belle et intéressante que l'arrivée ». On arrive tous exactement au même point peu importe ce que l'on fait. Ce qui reste, c'est ce qu'on a construit pendant ces 4 ans. Profitez en donc, personnellement, techniquement, ou les deux. Chacun a son style. Le trajet fait la différence. C'est cela qui compte.

Tout d'abord j'aimerais remercier tous ceux qui au Brésil, malgré nos politiciens, croient que la vraie démocratie, que la vraie justice sociale, que l'indépendance et le développement qui font d'un Pays une Nation, passe par l'éducation et par la culture. Je remercie donc la CAPES-COFECUB (Coordenação e Aperfeiçoamento de Pessoal de Ensino Superior) pour m'avoir soutenu financièrement pendant le développement de ce travail. Je remercie spécialement Marta Elias de la CAPES pour sa compétence et sa disponibilité.

Je tiens à remercier Roland Balter pour m'avoir fait l'honneur de présider le jury de ma soutenance, Xavier Rousset de Pina pour sa participation comme membre de jury et pour les remarques faites sur le manuscrit. Je remercie encore Jean-Louis Pazat et Jean-François Méhaut/Jean-Marc Geib pour avoir, malgré leurs emplois du temps très chargés, accepté de rapporter cette thèse. Vos remarques, vos avis, et vos commentaires ont été sans aucun doute très précieux.

Une des mes passions est la F-1, plus précisément la *Scuderia*. Nous sommes habitués à voir pendant le championnat un seul homme. Mais si on le voit, c'est parce que derrière lui il y a une vraie équipe qui lutte pour un seul objectif. Avec cette thèse je me sens un peu dans le rôle d'un pilote. Vous vous apprêtez à lire mon travail, mais sans une équipe ceci ne serait pas possible. J'aimerais donc remercier « mon » équipe.

Jacques Briat, mon « manager » et responsable de la « tactique de course ». Tu m'as beaucoup appris. Ça a été un vrai honneur d'avoir eu la possibilité de travailler avec toi.

Brigitte Plateau, la « patronne » de l'équipe, pour m'avoir accueilli au sein du projet Apache et pour avoir fait confiance à mes capacités. Grâce à cela j'ai pu évoluer dans les « circuits » du parallélisme.

Le labo est devenu pratiquement une deuxième maison, sans doute pour la bonne ambiance qui y regne. Je crains de citer des noms et d'oublier quelqu'un. Merci aux anciens thésards pour leur héritage, et aux nouveaux arrivants... bon courage. Merci aux personnels des moyens informatiques et du secrétariat. Merci également à tous les permanents (Yves : Forza, Ferrari !).

Quand même il ne serait pas juste de ne pas citer quelques noms...

Un merci spécial à Jean-Marc Vincent pour m'avoir prodigué les leçons de statistique et de « chrono », et à Jacques Chassin de Kergommeaux pour m'avoir donné la chance de participer aux coopérations internationales.

À Isabelle Lamadieu et à François Ottogalli pour la lecture et les corrections de la toute première version de ce manuscrit. La Fontaine sera toujours là. Merci aussi pour être là au moment où je « croisais la ligne de l'arrivée ». À Nicolas Maillard, à Jean-Guillaume Dumas, et à Olivier Briant pour la révision finale. À Olivier encore pour m'avoir fait partager sa connaissance en « graphismes ». À Andréa Charão et à Luciana Cavalheiro (sans elles il n'y aurait pas eu de pot).

Aux autres doublés de « pilotes » et rédacteurs : Mathias Doreille, Gregory Mounié, François Galilée, Alfredo Goldman, Olivier Briant et Gerson Cavalheiro. Ça a été un plaisir de courir avec vous à mes côtés.

Mon merci spécial au grand « pilote » `#man Benhur Stein` avec qui j'ai eu le plaisir de partager le bureau pendant 4 ans. Merci pour toutes les fois où tu as joué le « mécano », pour toutes les discussions techniques et philosophiques. J'ai beaucoup appris. Les longs et pénibles « tours » de rédaction ont pris une allure différente avec toi.

Merci à mes amis Gerson Cavalheiro, Benhur Stein, Vanderlei Rodrigues et Fabiana Z. Wilke pour leur soutien principalement aux derniers tours. Votre support a été fondamental pour que je tienne la route.

Au « Maître » Philippe O. Navaux pour mes premiers pas dans la recherche. Aux « Trois Mousquetaires », Celso M. da Costa, Eric Morel et Eddine Kannat. Si cette thèse existe vous avez aussi votre parcelle de faute.

À Pierre, Simone, Virginie et Isa Lamadieu pour l'« adoption ». Je vous serai éternellement redevable pour tout... Molière, le foot, Maupassant, le vélo, Racine, le ski, l'apprentissage de la culture française et celle du midi, les vins, les repas (désolé, mais je ne mange toujours pas du fois gras), la « chasse » aux myrtilles ou aux châtaignes. La liste est longue... et tout ce que je trouve à dire c'est « Merci ».

À mes frères André, Leandro et Leonardo. Particulièrement André et Leonardo pour avoir géré « mes affaires » au Brésil, par tous les e-mails qu'on a pu échanger pendant 4 ans, et pour le voyage avec l'*Enterprise*.

Mes remerciements à une grande femme, Nelcy, celle qui m'a appris la joie de vivre et le vrai sens des mots force et courage. Merci Maman pour ton dévouement.

Grazie al più bravo di tutti gli insegnanti che ho mai conosciuto, quello che un giorno mi ha detto che la più grande eredità che mi lasciava era la mia educazione. Oggi che ho preso una parte più grande di questa eredità ti rendo omaggio. Io ti ringrazio anche per avermi fatto vedere e scoprire il mondo guardando sempre avanti. Babbo, grazie ! ![1]

ET... À TOI ! !

[1]Merci au meilleur Professeur que j'ai jamais connu. Merci à celui qui un jour m'a dit que le plus grand héritage qu'il me laissait c'était mon éducation. Maintenant que je viens de prendre une tranche de plus de cet héritage, je te rends mon hommage. Merci aussi pour m'avoir fait voir et découvrir le monde en le regardant toujours de face. Papa, je te remercie ! !

Table des matières

Table des figures

Liste des tableaux

Introduction

« Rien ne sert de courir ; il faut partir à point »
Jean de La Fontaine, Le lièvre et la tortue

Depuis quelques années, l'extension de l'utilisation de l'informatique dans les différents domaines de recherche scientifique et technique se traduit par un besoin croissant de puissance de calcul. La croissance de la puissance de calcul des microprocesseurs n'est plus suffisante pour couvrir ce besoin. L'assemblage de multiprocesseurs à l'aide de réseaux haut débit est à même de fournir la puissance nécessaire. Toutefois un problème technologique persiste : comment exprimer la parallélisation d'un calcul et exploiter de façon efficace le potentiel offert par ces nouvelles machines ? La **multiprogrammation légère** et **la programmation par échange de messages**, sont des éléments nécessaires.

La **multiprogrammation légère** est de plus en plus utilisée pour exploiter un **parallélisme à grain fin** présent dans une classe d'applications dites **irrégulières**. La **multiprogrammation légère** permet d'exploiter efficacement un parallélisme réel offert par des machines multiprocesseurs à mémoire partagée. La création d'un processus léger est une opération moins coûteuse que celle d'un processus « classique » de type UNIX. De plus grâce au partage de l'espace d'adressage, les processus légers peuvent communiquer de façon très efficace entre eux. La coordination entre les activités est garantie par des opérateurs de synchronisation tels que les verrous, les sémaphores et les variables de condition.

En outre, l'accroissement d'efficacité des réseaux d'interconnexion nous permet de réaliser des machines parallèles à mémoire distribuée de faible coût en rassemblant des stations de travail « classiques » : les grappes de stations. Le modèle de programmation qui s'impose ici est celui de **la programmation par échange de messages**. Les processus communiquent en échangeant des messages à travers un réseau d'interconnexion. La mise en œuvre de ce modèle se fait en terme de processus « lourd ». Elle est adequate pour des applications qui présentent un **parallélisme à gros grain** et une bonne localité de données.

L'utilisation de **grappes de stations multiprocesseurs** peut nous permettre l'exploitation à la fois d'un parallélisme *à grain fin* (interne à un processus offert par les processus légers) ainsi qu'à *gros grain* (entre les différents processus qui

coopèrent à la résolution du problème). L'exploitation simultanée de ces deux modèles exige la présence de mécanismes de communication entre les processus légers qui ne partagent pas le même espace d'adressage. Plusieurs environnements de programmation parallèles, dont certains seront présentés au cours de cette thèse, ont été proposés pour combler ce besoin.

Le projet APACHE a conçu et réalisé l'environnement de programmation parallèle ATHAPASCAN pour le développement de ses applications parallèles. La conception d'un tel environnement est guidée par un compromis entre trois points essentiels : les **fonctionnalités fournies**, la **portabilité**, et l'**efficacité** des mécanismes mis en place. Il est très difficile de privilégier ces trois points simultanément. Il faut trouver un bon équilibre entre ces points.

L'environnement ATHAPASCAN repose sur un modèle de tâches et objets partagés. Cette abstraction est fournie par une interface de programmation de plus haut niveau : ATHAPASCAN-1. Elle inclut des politiques de placement automatique, et se présente comme une bibliothèque générique C++. ATHAPASCAN-0 est le noyau exécutif de l'environnement de programmation parallèle ATHAPASCAN, la portabilité est donc fondamentale. Pour ceci, ATHAPASCAN-0 est basé sur des standards largement répandus sur une grande gamme de systèmes comme **POSIX threads** et **MPI**.

En ce qui concerne l'efficacité, ATHAPASCAN-0 hérite de celle offerte par les architectures cibles puisque le choix de conception repose sur l'emploi de « produits standards ». Cependant les surcoûts introduits par les primitives d'ATHAPASCAN-0, comparés aux primitives « natives », ne sont pas très importants, ce que nous montrons plus tard (chapitre 6).

Les fonctionnalités fournies par ATHAPASCAN-0 permettent de « voir » les applications comme un **réseau dynamique de processus légers communicants**. Essentiellement, ATHAPASCAN-0 offre des primitives « classiques » pour l'emploi de processus légers et des communications point à point. Ces primitives sont enrichies par la capacité de création de processus légers à distance et par la communication entre processus légers appartenant à différents processus.

Objectifs de la thèse

Mon travail de thèse se place dans le contexte du développement du noyau exécutif ATHAPASCAN-0. Il porte plus précisément sur l'évaluation et le réglage de ce noyau sur des grappes de multiprocesseurs de type SMP[2].

L'intérêt d'un noyau associant processus légers et communication est d'une part d'explorer le parallélisme multiprocesseur et de permettre le recouvrement des délais de transit sur le réseau par du calcul. La technique utilisée est la multiprogrammation. La mise en œuvre de ce mariage repose sur la capacité à dé-

[2]Symmetric MultiProcessors

tecter la possibilité d'émettre sur le réseau, ou la nécessité de recevoir du réseau, de façon à faire avancer au mieux les communications en minimisant les surcoûts de cette multiprogrammation. Comme l'architecture cible et le système d'exploitation ne permettent pas toujours un contrôle précis, il est alors nécessaire de trouver le meilleur compromis. Mon travail de thèse a porté sur l'évaluation du fonctionnement d'ATHAPASCAN (parallélisme de calcul, parallélisme de communication, recouvrement) et son réglage pour des grappes de multiprocesseurs SMP.

Ce réglage a nécessité des modifications importantes du code initial d'ATHAPASCAN. En effet, la version initiale avait été développée pour une grappe de monoprocesseurs (IBM SP1) et les problèmes d'efficacité (conflit sur verrous, conflits de caches, etc) ne sont apparus que lors du passage sur multiprocesseur. Un cycle d'étude des problèmes spécifiques aux multiprocesseurs, des modifications, d'évaluation a donc été nécessaire pour atteindre un fonctionnement efficace sur une telle architecture.

Organisation du document

Dans le premier chapitre « **Le domaine de recherche** », nous détaillerons le contexte de ce travail et le projet APACHE. Nous nous positionons aussi dans le domaine de l'exploitation du parallélisme. Nous présenterons les concepts généraux utilisés tout au long de ce document tels que grain de parallélisme, algorithmes réguliers et irréguliers, types d'architectures de machines parallèles, différents paradigmes de programmation, etc. Nous montrerons l'intérêt d'un support d'exécution pour l'exploitation efficace des applications parallèles dites irrégulières. Pour conclure ce chapitre nous présenterons l'évolution de l'environnement de programmation ATHAPASCAN-0, de son prototype initial jusqu'à l'état actuel, en montrant comment ATHAPASCAN a dû s'adapter aux évolutions des architectures de machines et des systèmes d'exploitation.

Le second chapitre, « **La multiprogrammation légère** », est une introduction au concept de processus légers (*threads*). Nous y discuterons les différents modèles pour leur implantation tout en commentant les avantages et les inconvénients de chacun de ces modèles. Nous présenterons l'intérêt de l'emploi de la multiprogrammation légère. Son utilisation croissante depuis le début des années quatre-vingt-dix a provoqué, par souci de portabilité, la standardisation d'une interface : le standard POSIX ANSI/IEEE 10003.1, aussi appelé ***Pthreads***, ou encore **POSIX.1c *threads***. Ce chapitre se termine sur les aspects liés à l'évaluation et à la performance d'implantation de processus légers.

Le troisième chapitre, « **MPI : Message Passing Interface** », est dédié à un autre paradigme de programmation largement répandu sur les architectures distribuées : celui de la programmation par échange de messages. Nous décrirons donc le type de parallélisme exploité via ce paradigme. Ensuite nous introduisons le standard MPI qui définit une interface et les fonctionnalités de base pour une

bibliothèque de communication. Ce standard, dans sa version originale, ne définit pas une série d'aspects liés à la programmation parallèle, particulièrement ceux attachés à l'utilisation conjointe de MPI et des processus légers. Certains aspects ont été pris en compte lors de la définition de MPI-2. Nous ferons un survol de ces évolutions. Nous présenterons aussi la notion de recouvrement calcul-communication. Nous conclurons par la présentation d'une suite de jeux de tests (*benchmark*) pour l'évaluation de bibliothèques de systèmes de communication. Ces jeux de tests sont basés sur le travail du consortium PARKBENCH.

Le quatrième chapitre, « **L'intégration de *threads* et communications** », décrit plusieurs aspects et modèles liés au problème d'intégration des *threads* et communications. Initialement nous discuterons les modèles utilisées pour intégrer les communications au calcul, comme les **messages actifs**, le modèle **upcall**, et le modèle **popup**. Tous ces modèles font face au problème de la détection de l'arrivée des messages sur le réseau et au choix d'une technique appropriée d'acquisition : l'**interruption** versus la **scrutation**. Nous y introduirons une des problématiques traitées par cette thèse : celle de « **tester et faire avancer les communications** ». Nous conclurons ce chapitre par état de l'art sur les principaux environnements de programmation distribuée basés sur les processus légers et sur la communication. Une attention particulière sera portée sur la façon de traiter le problème de la scrutation réseau.

Le cinquième chapitre, « **ATHAPASCAN-0 sur multiprocesseurs** », décrit notre implantation d'ATHAPASCAN sur des machines multiprocesseurs SMP. Tout d'abord nous introduirons les principaux concepts liés à cette classe de machine et leur influence sur l'efficacité d'application. ATHAPASCAN-0, comme plusieurs autres systèmes, a été développé pour une architecture monoprocesseur. Certaines décisions d'implantation ont été prises en supposant une telle architecture comme cible. Nous montrerons que cela a provoqué des conséquences néfastes au niveau de la performance lorsque cette version a été portée directement sur une machine multiprocesseurs. À partir de cela nous montrerons les changements que nous avons apportés à ATHAPASCAN-0 pour qu'il soit exécuté de façon efficace sur cette classe de machines.

Le sixième chapitre, « **Évaluation de performance** », est l'évaluation du noyau exécutif ATHAPASCAN-0 sous plusieurs aspects. Nous commencerons en analysant le comportement d'ATHAPASCAN (surcoûts) par rapport aux bibliothèques de base : processus légers et communication. Ensuite, en utilisant des programmes synthétiques nous allons vérifier le comportement d'ATHAPASCAN-0 à différents régimes d'opération. Nous réaliserons essentiellement des mesures en jouant sur le rapport communication et calcul des applications de test.

Finalement, le dernier chapitre, « **Conclusion** », fait le point des résultats obtenus par cette thèse et décrit quelques évolutions futures du noyau exécutif ATHAPASCAN-0.

Tout au long de ce document, dans un souci de clarté nous préférerons l'emploi

de certains termes techniques en anglais, principalement lorsque nous parlerons de la **multiprogrammation légère**. Pour le terme *thread*, en effet, il existe des traductions en français, comme fils d'exécution, flot d'exécution et processus légers. Cependant, nous pensons que l'emploi de ces termes peut parfois rendre difficile la présentation, principalement lorsque nous parlerons des processus et de leurs relations de filiation. Le terme « flot », à notre avis, est moins employé dans la littérature en général pour exprimer la notion de *thread* et peut aussi prêter à confusion avec « flot de données » dans le cadre des communications. Les termes anglais seront toujours employés *en italique*.

1

Le domaine de recherche

« Travaillez, prenez de la peine, c'est le fond qui manque le moins »
Jean de La Fontaine, Le laboureur et ses enfants

Dans ce chapitre nous présenterons une classification couramment employée pour les architectures de machines parallèles et les différents paradigmes et modèles de programmation disponibles. Ensuite nous introduirons les concepts généraux utilisés tout au long de ce document tels que le grain et la régularité. Nous allons montrer l'intérêt d'avoir un support exécutif pour la réalisation d'applications parallèles irrégulières. Nous nous positionnerons dans le domaine de l'exploitation du parallélisme et nous discuterons du projet APACHE en présentant ses différents axes de recherche. Pour conclure, nous présenterons l'évolution du noyau exécutif ATHAPASCAN-0 depuis son prototype initial jusqu'à l'état actuel en montrant comment ATHAPASCAN a su s'adapter aux évolutions de l'architecture des machines et des systèmes d'exploitation.

1.1 Les architectures de machines parallèles

Pendant longtemps la classification proposée par Flynn [71] dans les années 70 a été utilisée pour caractériser les différents types d'architecture d'ordinateurs. Cette classification est fondée sur le nombre de flux de données (Single Data stream ou Multiple Data stream) et sur le nombre de flux d'instructions (Single Instruction stream ou Multiple Instruction stream) présents sur une architecture donnée. De la combinaison de ces possibilités résulte quatre catégories :

SISD (Single Instruction Single Data) : C'est le modèle de fonctionnement séquentiel qui correspond à l'architecture dite « de Von Neuman » et concerne la majorité des ordinateurs commercialisés actuellement. Cette architecture est organisée autour de deux éléments : la mémoire et le processeur. Le processeur lit une à une les instructions de la mémoire et les effectue sur un flux de données provenant de la mémoire.

SIMD (Single Instruction Multiple Data) : Dans ce modèle les unités de traitement de données sont dupliquées et une seule unité de contrôle les commande. Ceci implique qu'à un instant donné, les unités de traitement exécutent une même instruction sur des données différentes. Ces exemples de machines appartiennent à la catégorie des processeurs matriciels (*array processors*).

MISD (Multiple Instruction Single Data) : Dans ce modèle un ensemble d'instructions est exécuté sur un seul flux de données. Certains auteurs considèrent qu'aucune architecture ne correspond à ce modèle, d'autres considèrent les architectures de type *pipeline* comme représentantes de cette catégorie.

MIMD (Multiple Instruction Multiple Data) : Cette architecture consiste à repliquer le processeur SIMD. Chaque processeur est libre d'exécuter les instructions qui lui sont propres sur un flux de données qui lui est propre. On a donc différents flux de contrôles sur différents flux de donnés. Les machines multiprocesseurs sont des exemples typiques de ce modèle.

Du fait de ces critères trop généraux, la classification de Flynn devient inadéquate pour les systèmes distribués car tous sont considérés comme appartenants à la classe MIMD quelles qui soient leurs différences. Aujourd'hui il est courant [153][162] de subdiviser la classe MIMD en deux groupes : les **multiordinateurs** et les **multiprocesseurs**. Le type d'interconnexion utilisé pour relier mémoires et processeurs , bus ou commutateur (*switch*), génère d'autres sous-divisions pour chacun de ces groupes. La figure 1.1 montre cette nouvelle taxinomie.

Les multiprocesseurs, aussi appelés systèmes fortement couplés (*tightly coupled sysems*), ou encore machines à mémoire partagée, sont caractérisés par plusieurs processeurs partageant un même espace d'adressage. Par rapport au type d'accès que les processeurs font aux mémoires, ces machines sont encore sous divisées en machines **UMA** (*Uniform Memory Access*) ou **NUMA** (*Non-Uniform Memory Access*).

L'architecture UMA reste la plus courante, les processeurs, les mémoires et les dispositifs d'entrée/sortie sont connectés à un même bus. Le principal avantage de cette architecture est qu'elle est simple à réaliser. Mais elle présente un problème de « scalabilité » (*scalability*). En augmentant le nombre de processeurs, nous augmentons aussi la contention d'accès au réseau d'interconnexion processeur-mémoire. Pour limiter cette charge, on utilise des mémoires caches ce qui nécessite la mise en place d'un protocole de maintien de cohérence des caches. Ces difficultés limitent la

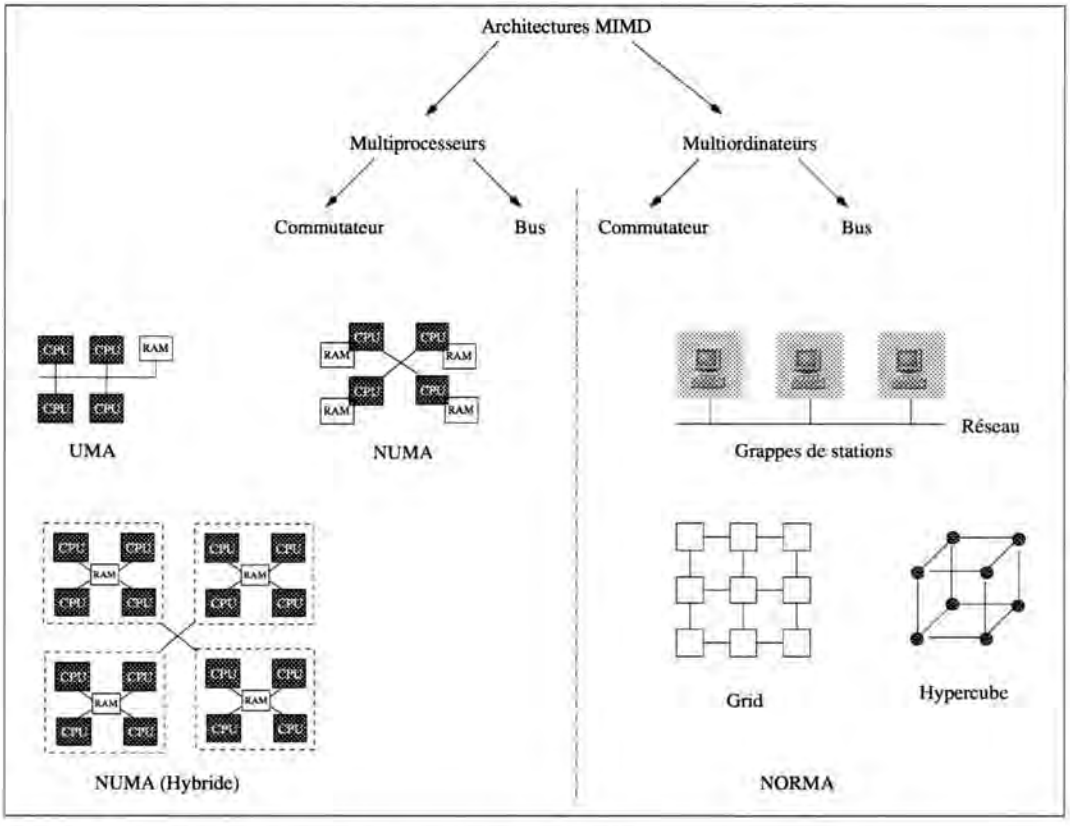

Figure 1.1 *Taxinomie pour les architectures MIMD*

taille des machines UMA, typiquement, à des machines de huit à seize processeurs. Sur le marché à l'heure actuelle on retrouve plusieurs machines utilisant cette architecture, nous pouvons citer comme exemple, parmi d'autres, SUN Enterprise (300-6000), DELL PoweEdge, SGI Challenge/Onyx, SGI octane.

Dans un multiprocesseur NUMA chaque processeur possède une mémoire locale qui forme avec les mémoires des autres processeurs un seul espace d'adressage. Les accès aux positions mémoire qui correspondent aux mémoires distantes (celles appartenant à un autre processeur) sont plus lents qu'un accès à la mémoire locale, d'où la référence à « accès mémoire non uniforme ». Ici aussi, il est nécessaire d'introduire des caches pour assurer une efficacité raisonnable aux accès mémoires. Ces machines sont appelées **cc-NUMA** où le « cc » » veut dire « cache cohérence ». Il est encore possible d'organiser les machines NUMA de façon hybride, c'est à dire, qu'un groupe de processeurs accèdent à une mémoire locale de façon uniforme, et accèdent à distance à la mémoire appartenant à un autre groupe de processeurs. L'abandon de la contrainte de temps uniforme d'accès à la mémoire permet de construire des machines à plus grand nombre de processeurs. Dans ce cas le programmeur devra soigneusement organiser ses données en mémoire de façon à limiter l'effet des accès non locaux. Comme exemple de cette classe de machines citons ici la machine Origin 2000 (SGI). Un dernier type d'architecture multiprocesseur est l'architecture **COMA** (*Cache Only Memory Architecture*), où les mémoires caches remplacent le

rôle des mémoires vives sur l'ensemble de processeurs. Elles présentent les mêmes problèmes de programmation efficace.

Le couplage fort pour la mémoire introduit des contraintes importantes sur le réseau d'interconnexion processeur-mémoire et nécessite la mise en place de protocole de gestion de cohérence de caches : c'est pourquoi le passage à un plus grand nombre de processeurs a nécessité un couplage plus lâche. Les architectures multiordinateurs suivent cette philosophie.

Dans le groupe des **multiordinateurs**, ou **systèmes faiblement couplés** (*loosely coupled system*), les nœuds qui définissent la machine parallèle sont indépendants les uns des autres, c'est à dire, que chaque nœud est composé par un processeur et par une mémoire locale. Le nœud, afin d'accéder à une mémoire distante (celle appartenant à un autre nœud) le fait de façon explicite et conventionnelle par le biais d'échange de messages avec l'autre nœud. Cette architecture, ne permettant pas un accès direct à une mémoire distante, est aussi connue sous l'appellation **NORMA** (*NOn Remote Memory Access*) ou encore, **machines à mémoire distribuée**. Les grappes de stations (*NOW - network of workstations* ou CLUMPS *CLuster of MultiProcessorS*) et la machine IBM SP2, par example, sont représentatives de cette classe. Pour les grappes de stations, du point de vue matériel, le système peut être dit hétérogène ou homogène. Une grappe de stations est dite hétérogène quand les nœuds qui la composent sont des machines différentes entre elles au niveau du processeur.

Outre l'aspect matériel, nous avons aussi l'aspect **système d'exploitation**. Pour les multiordinateurs, étant donné qu'ils peuvent être obtenus en assemblant différentes machines « classiques » sur un réseau, chaque machine est indépendante et peut avoir son propre système d'exploitation. Comme dans le cas matériel, les machines peuvent être homogènes, si la grappe est composée par des machines de même type de système d'exploitation, ou hétérogène, dans le cas contraire.

Les multiprocesseurs ont forcément le même système d'exploitation, toutefois celui-ci peut être conçu de trois façons différents : **maître-esclave**, **fonctionnellement asymétrique** et **symétrique**. Dans l'organisation **maître-esclave**, un processeur est responsable, par exemple, du traitement des entrées/sorties et des interruptions. Ou, bien encore, le maître est le seul à exécuter le code du noyau et les processeurs esclaves exécutent seulement le code d'utilisateur. Dans un **système fonctionnellement asymétrique**, différents sous systèmes sont attribués à différents processeurs en créant une spécialisation de tâches par processeur. Enfin, dans les **systèmes symétriques** (*SMP - Symmetric MultiProcessing*), tous les processeurs exécutent le code noyau, les code utilisateurs, et sont en compétition pour les ressources systèmes (mémoire et entrée/sortie) de façon égale. Cette approche est la plus courante dans les systèmes multiprocesseurs actuels.

1.2 Modèles de programmation parallèle

Un modèle de programmation représente l'abstraction d'une machine dans le but de séparer les aspects concernant le développement de l'application de ceux liés à son exécution effective. Des modèles existent à différents niveaux d'abstraction mais, pour être utiles, ils doivent satisfaire un certain nombre de propriétés dans différents domaines [143] :

Fonctionnalités : le modèle doit abstraire le plus possible les traits liés à l'organisation d'un programme parallèle, comme la décomposition, l'ordonnancement, les communications et la synchronisation. La décomposition est la façon par laquelle un programme parallèle est divisé en tâches indépendantes. L'ordonnancement est lié à l'attribution de ces tâches aux processeurs. Ces deux items sont particulièrement importants pour obtenir l'efficacité optimale d'exécution d'un programme parallèle. La communication est une conséquence des dépendances de données entre deux tâches, et nécessite la détermination des points adéquats où réaliser l'échange de données. Enfin, la synchronisation permet de garantir que l'état global des tâches reste cohérent c'est à dire que les propriétés invariantes d'état restent toujours vérifiées. Une bonne abstraction de ces traits aide à la compréhension et au développement d'applications parallèles.

Méthode de développement : comme précédemment indiqué, le développement d'une application parallèle repose sur une identification précise des dépendances de données et des invariants d'état. Il est important que les opérateurs implantant le modèle aient une sémantique claire et précise de façon à faciliter une prédiction comme une analyse du comportement de l'application.

Simple compréhension : le modèle de programmation doit être assez robuste pour abstraire les complexités de la programmation parallèle et qu'il soit en même temps facile à comprendre, sans perdre pour autant ses capacités d'expression.

Indépendance de l'architecture : un modèle doit être indépendant de l'architecture de la machine pour garantir une portabilité et une pérennité des applications développées. Cette abstraction doit permettre une utilisation efficace des architectures cibles. Des traits matériels différents qui ne peuvent pas être abstraits efficacement en un concept unique doivent donc faire l'objet de concepts différents.

Implantation efficace : un modèle doit pouvoir être implanté de façon efficace sur la plupart des architectures « conventionnelles ».

Modélisation de coût : la prédiction des performances d'une application parallèle et la détermination des ressources nécessaires à son exécution exigent des moyens d'estimer le coût du modèle et de ses opérateurs. Les coûts effectifs pour une architecture doivent pouvoir se calculer à partir de ce modèle de

coût et des paramètres physiques de la machine.

En fonction de ces critères les modèles peuvent être classés, en ordre décroissant d'abstraction, en six catégories selon la façon dont le parallélisme doit être explicité [144] : parallélisme implicite, parallélisme explicite, décomposition explicite, placement explicite, guidée par événement, et finalement, totalement explicite.

Dans un modèle à **parallélisme implicite**, toutes les activités liées à l'exécution parallèle sont complètement cachées au programmeur. L'approche la plus commune est de décrire les calculs d'une manière déclarative par le biais d'un ensemble de fonctions et/ou d'équations. La parallélisation est obtenue par une étape de traduction de cette description vers un programme parallèle par des **compilateurs paralléliseurs** [67][10][127]. Les langages de programmation fonctionnelle sont représentatifs de cette catégorie, ainsi que les langages de programmation logique comme PROLOG [66].

La deuxième catégorie de modèles à **parallélisme explicite**, est celle où le parallélisme est décrit de façon explicite sans pourtant définir la façon dont l'application est divisée en tâches, ni le placement (ordonnancement) de celles ci ni leurs communications. Comme exemples de cette catégorie nous avons les langages *dataflow* et les langages de type *concurrent logic* (PARLOG [45], Multilisp [93], Concurrent Prolog [139], GHC [159], etc).

Le modèle à **décomposition explicite** considère que les langages offrent des primitives explicites pour définir les tâches à exécuter en parallèle. Cependant, la réalisation du placement (ordonnancement) et des communications reste toujours cachée au programmeur. Deux modèles largement répandus font partie de cette troisième catégorie : le modèle LogP [51] et le modèle BSP [115]. ATHAPASCAN-1 [82], Cilk [23] et Jade [109] sont aussi des exemples.

Dans un modèle de programmation à **placement explicite**, les programmeurs accomplissent la décomposition en différentes tâches et réalisent leur placement sur les processeurs. Par contre communications et synchronisation sont assurées de façon transparente. Les langages de coordination (*coordination languages*) qui séparent les aspects de calcul des aspects de communication, comme Linda [4], en sont un exemple. D'autres exemples de langages de programmation appartenant à ce modèle sont PCN [77], Compositional C++ [37], ou encore des languages graphiques tels que Code [122] ou Parsec [68].

Naturellement, les modèles précédents sont extrêmement difficiles à implanter. Les modèles suivants proposeront différentes façons d'assurer explicitement les synchronisations et mouvements de données d'un programme parallèle.

Le modèle **guidée par événements** restreint les mouvements de données et les synchronisations à se faire via la « production d'un événement » déclenchant des traitements comme conséquence. Deux exemples bien connus de ce modèle sont les actors [2][3] et les messages actifs [166]. Ces deux systèmes ont en commun le fait qu'ils réduissent la communication à l'envoi de messages asynchrones. Ces modèles simples présentent un problème de définition de comportement correct rel-

ativement au contrôle de flux des événements. Deux sémantiques correctes ont été proposées. La première est synchrone et n'est pas appropriée à une exécution parallèle (ESTEREL[16], LUSTRE[92], par exemple). La seconde suppose une capacité infinie de mémorisation des événements-action à exécuter (par exemple, la réception et stockage de message par des couches protocoles réseau).

Finalement, le dernier modèle, **totalement explicite** car tous les aspects de décomposition,de placement, de communication et de synchronisation sont tous à charge du programmeur de l'application. Si une machine abstraite pour ce modèle est relativement facile à implanter, sa programmation est difficile car tous les détails de la parallélisation sont apparents. Il existe aujourd'hui plusieurs modèles de ce type accompagnés de leur paradigme de programmation. Les quatre les plus répandus sont : modèle à mémoire partagée, modèle à passage de message, modèle à communication par rendez vous, et modèle à appel de procédure à distance (RPC).

La programmation par **mémoire partagée** est une excellente option pour utiliser les architectures multiprocesseurs. La communication est faite naturellement par des variables partagées et par sémaphores (synchronisation). La plupart des systèmes d'exploitation mettent à disposition une interface de programmation implantant ce modèle. Un des avantages de ce modèle est que la décomposition et le placement sont automatiquement traités par le système d'exploitation. Cependant, il est trop lié à un style d'architecture de machine, ce qui réduit sa portabilité.

L'**échange de messages** est la technologie de base qui vise à réaliser des communications entre les processeurs d'une architectures MIMD. Ce paradigme sont basés sur deux primitives très simples, *send* et *receive*, dont les paramètres sont l'identification de l'autre processeur impliqué à la communication, l'adresse du message à envoyer (ou l'adresse pour stocker le message à la réception). Ce modèle, essentiellement le même pour n'importe quelle machine de type MIMD, a donné de nombreuses variantes. Afin d'assurer la portabilité et la facilité d'utilisation (et de réutilisation) des programmes un standard pour l'interface de programmation par échange de messages a été élaboré : MPI. Ici aussi le problème du contrôle de flux des messages est critique. Deux sémantiques correctes ont été proposées. La première est synchrone (rendez-vous) et la seconde asynchrone (canal non borné).

Dans une communication avec **rendez-vous**, émetteur et récepteur doivent demander à communiquer pour que la transmission ait lieu. OCCAM[103] est une implantation d'un tel modèle. Dans le modèle à canal non borné, l'émetteur peut émettre, le message sera mémorisé et délivré au récepteur quand celui-ci le demandera. La première solution (synchrone) ne permet pas d'utiliser commodement toute la bande passante du réseau. La seconde (asynchrone) suppose une capacité non borné du réseau. La plupart des bibliothèques de communication offre un modèle à canal borné à charge pour le programmeur de calculer la taille des tampons nécessaires à son application pour éviter les blocages.

L'**appel de procédure à distance** (*Remote Procédure Call* - RPC) [21] étend le mécanisme « conventionnel » d'appel de procédure des langage comme C. Un RPC

est un appel de procédure entre deux différents processeur, l'appelant et l'appelé. Quand un processus du processeur A réalise un appel vers une procédure à distance sur un autre processeur B, le processus de B reçoit les paramètres d'entrée, exécute la procédure, et puis renvoie les résultats vers l'appelant (A). Comme le rendez-vous, le RPC est synchrone. Quand le processus de A réalise un appel à distance d'une procédure P, il reste bloqué jusqu'au moment où la procédure P est finie et le résultat retourné. Ceci limite le dégré de parallélisme. Différents façons d'exécuter la procédure appelée sont possibles. Un processus dit serveur peut exécuter ces appels de procédure l'un après l'autre. Une autre solution consiste à créer un tel processus pour chaque appel. Les langages DP [29], Cedar [151] et concurrent CLU [47] sont basés sur ce mécanisme.

On peut voir ce modèle comme une extension du modèle guidé par événement. Comme le modèle par événement, ou le modèle à échange de message, il doit faire face au problème du contrôle de flux des instances de procédures. Le seul modèle existant suppose une capacité non bornée de mémorisation des invocations. Naturellement, les implantation offrent une capacité bornée que le programmeur doit fixer. ADA[119] implante un modèle mixte où l'invocation suit un modèle d'appel de procédure et où le traitement est fait par un processus serveur. La communication des arguments et résultats se fait par rendez-vous.

1.3 Applications régulières et irrégulières

Un programme parallèle peut être vu comme un ensemble de tâches qui exécutent une séquence de calculs et éventuellement des phases de communication pour échanger des données entre elles. Le coût de la communication joue un rôle très important dans l'obtention d'une exécution efficace. En effet, la communication est nécessaire dès qu'un calcul séquentiel est parallélisé. Il ne faut pas que ce surcoût de communication annule le gain de parallélisation. Plus ce surcoût est faible, plus le calcul élémentaire « rentable » peut être de « petit » grain.

Certaines taxinomies [134] classent les algorithmes en fonction de la relation entre la complexité de calcul (« poids ») et la complexité de la communication. Il est donc envisageable qu'un programme parallèle offre une relation calcul/communication bien adaptée à la machine cible. Savoir si une application est adaptée à une machine suppose la connaissance préalable du comportement de l'application en termes de phases de calcul et de communication et leurs durées. Cette connaissance se représente par un graphe de dépendance dont les arêtes sont étiquetés par des coûts. En fonction du graphe de dépendance les algorithmes parallèles sont classés en **réguliers** et **irréguliers** [83][61].

Un algorithme régulier est celui dont le comportement est connu avant exécution ou qui peut se prédire à chaque étape d'exécution. Cela nous permet d'organiser un programme parallèle en tâches de « poids » équivalents et composées de façon à minimiser les communications. Un algorithme irrégulier est celui pour lequel on

ne peut prédire le comportement qu'au moment de son exécution, ou pour lequel cette prédiction nécessiterait un calcul de l'état global de l'application, ou encore ce comportement présente une grande variabilité. Ces algorithmes sont caractérisés par la création dynamique d'un nombre important de tâches de grain très diverses car le type de découpage du programme parallèle engendre des sous problèmes de taille variable. L'ordonnancement de ces sous-problèmes de façon équitable, pour éviter les déséquilibres de charge sur les différents nœuds participants à l'exécution, n'est pas trivial.

En effet, la définition d'irrégularité d'un algorithme donné en [61] est liée exactement au critère d'ordonnancement : « *l'irrégularité d'un algorithme est directement liée à la complexité d'ordonnancement des tâches qui le composent* ». En d'autres termes, l'irrégularité d'un algorithme est intrinsèquement liée à la possibilité de construire, totalement ou partiellement, le graphe de précédence de son exécution.

1.4 Vers un support exécutif pour les applications irrégulières pour grappe de multiprocesseurs

La programmation parallèle distribuée est fondée sur l'implantation de la notion de processus et de mécanismes de communication. Initialement les systèmes ont offert des mécanismes permettant l'exploitation du parallèlisme des multiprocesseurs et du recouvrement calcul/entrée-sortie. Ces fonctionnalités IPC (*Inter Process Communication*) comprenait des outils de communication, les canaux (*pipes*), les canaux dédiés (*named pipes*), les messages (*messages queues*), et des outils de synchronisation et partage de mémoire. La communication entre ordinateurs se fait par une interface générique (sockets) et les protocoles de l'INTERNET (TCP/IP).

Ces mécanismes ont servi de base au développement des premiers environnements de programmation parallèles dans le but d'offrir au programmeur une abstraction et une expression du parallélisme plus souple. Les environnements de programmation les plus répandus sont ceux basés sur l'envoi de messages (MPI[147], PVM[85], etc) et sur le partage de la mémoire (Linda [4], DREAM [60], etc).

L'inconvénient de cette approche initiale est qu'elle est basée sur une implantation « lourde » à base de processus (processus UNIX). Un processus est une entité gérée par le système d'exploitation qui répresente l'exécution d'un fichier. Le terme « lourd » est employé pour désigner de telles entités car ils consomment des ressources très importantes du système comme les descripteurs, la mémoire, etc. Dans la mesure où les processus sont considérés comme indépendants ils nécessitent aussi la mise en œuvre de mécanismes de protection et isolation interdisant tout partage simple de zones de mémoire. Ces caractéristiques imposent des contraintes

à leur utilisation lorsqu'on développe une application parallèle :

- puisque les processus consomment des ressources systèmes, il est impossible à une application utilisateur de créer un nombre important de processus par machine ;
- la gestion des processus (création, changements de contexte, terminaison) présente des surcoûts importants du fait des contraintes de protection entre processus ;
- de même, les mécanismes de communication (IPC) sont « lourds » du fait des contrôles de protection entre processus.

Ces surcoûts sont assez forts pour interdire l'emploi d'environnements de programmation parallèles basés sur des processus « lourds » dans l'implantation d'algorithmes irréguliers. On est limité par le nombre de processus par application ; le grain de calcul des tâches tend à être fin ce qui rend intolérable les surcoûts de création et de communication entre processus. Il est évident que les implantations des modèles précédents sur les processus « lourds » sont inadéquats pour exploiter le parallélisme présent dans les algorithmes irréguliers. Il est nécessaire de disposer d'une implantation plus efficace du concept de processus : la multiprogrammation légère.

La multiprogrammation légère implante des processus « légers » comme unité d'exécution. Un processus légers est un flot d'exécution indépendant au sein d'un même espace d'adressage, c'est à dire, que les processus léger existent à l'intérieur de processus « lourds ». Un processus léger correspond à l'exécution d'une procédure d'un langage de programmation et à un fichier exécutable. La communication entre processus légers se fait naturellement par l'utilisation de variables partagées. Des mécanismes de synchronisation, comme verrous et variables de condition, sont disponibles pour gérer l'interaction entre les processus légers et le maintien de la cohérence lors des accès aux ressources partagées. Ceci se fait directement via la mémoire commune (les variables communes) à ces différents processus légers (procédures). Aucun recours aux mécanismes d'IPC n'est nécessaire. Initialement conçue pour recouvrir calcul et communication réseau dans les grands services INTERNET (et maintenant W3), la multiprogrammation légère est particulièrement adaptée à l'exploitation du parallélisme des multiprocesseurs.

La multiprogrammation légère est une technique incontournable. En effet, l'utilisation croissante des grappes de multiprocesseurs comme machines parallèles de faible coût, exige l'exploitation du parallélisme à deux niveaux. Au sein des nœuds de type multiprocesseur, les processus légers permettent d'exploiter le parallélisme entre les processeurs d'un même multiprocesseur. C'est le premier niveau de parallélisme. On parle aussi de parallélisme intra-nœud. Le deuxième niveau est celui d'un parallélisme inter-nœud, obtenu par l'exécution simultanée de différents processus sur différents nœuds. L'interaction entre ces processus est forcément faite

par des mécanismes d'échange de messages. La multiprogrammation légère permet de recouvrir les délais de communication par du calcul.

Un certain nombre d'environnements de programmation parallèle reposent simplement sur l'intégration d'un noyau de processus légers et d'un noyau de communication entre processeurs. Citons, parmi d'autres, Nexus[76], Chant[89], PM^2[120], TPVM[69], Panda[20], LPVM[168], Chare[140], etc. Ceci est aussi le cas du noyau exécutif ATHAPASCAN-0 [28][42][130] sur lequel porte mon travail de thèse.

Dans ces environnements simples, le placement des processus, les communications et synchronisations sont explicites. Ils implantent généralement un modèle « totalement explicite » (section 1.2) et on les qualifient de noyaux exécutifs RTS *Run Time Systems*). Ils servent de base pour l'implantation d'environnements plus évolués qui cherchent à automatiser les fonctions de placement de processus et de mouvement des données. C'est en particulier l'approche choisie par le projet APACHE pour la conception et le développement de l'environnement de programmation parallèle ATHAPASCAN.

1.5 Le projet APACHE

Le projet APACHE, acronyme pour **A**lgorithmique **P**arallèle et p**A**rtage de **CH**ar**E**, est un projet de recherche sur le calcul parallèle commencé au sein du Laboratoire de Modélisation et Calcul (LMC) appartenant à la confédération de l'Informatique et de Mathématiques Appliquées de Grenoble (IMAG), il est soutenu de manière conjointe par le CNRS, l'INPG, l'INRIA et l'UJF. Le domaine scientifique abordé par ce projet se déroule sur trois axes :

- développement d'un environnement de programmation parallèle à parallélisme explicite [130] c'est à dire à équilibrage de charge et mouvement de données automatiques ;
- algorithmique et développement d'applications parallèles.
- observation, évaluation et mise au point (débogage) de programmes parallèles.

Le projet APACHE propose une approche pour la programmation des machines parallèles pour le calcul haute performance offrant un bon compromis performance-portabilité. L'**environnement de programmation parallèle Athapascan**[1] est basé sur un modèle à parallélisme explicite accompagné d'un modèle de coût [34]. Il se présente comme une bibliothèque générique C++ dénommée **ATHAPASCAN-1** qui privilégie un modèle de parallélisme de tâches avec cohérence de données et permet une répartition automatique de la charge de calcul. Sa portabilité et son efficacité dépendent du noyau exécutif **ATHAPASCAN-0**. Celui ci est l'intégration d'un

[1] Athapascan est la langue des indiens Apaches

noyau de multiprogrammation légère et d'un noyau de communication. Des applications ont été réalisées sur ATHAPASCAN : dynamique moléculaire, chimie quantique, calcul formel, décomposition de domaine et simulation à événement discret. Enfin, un environnement de traces permet l'observation, l'évaluation et la visualisation d'Athapascan et de ses applications. Nous parlerons par la suite de chacun de ces composants.

1.5.1 L'environnement de programmation ATHAPASCAN

ATHAPASCAN est un environnement de programmation parallèle conçu pour être à la fois portable et efficace, et permettre de façon aisée le développement de programmes parallèles [28][42][130]. Pour atteindre ces objectifs l'environnement ATHAPASCAN est en fait organisé en trois niveaux : **ATHAPASCAN-1**, **ATHAPASCAN-0** et **Akernel**. Les primitives d'un niveau sont bâties en utilisant l'interface offerte par le niveau immédiatement inférieur. Cette organisation permet aussi le développement d'applications parallèles sur chacun de ses niveaux.

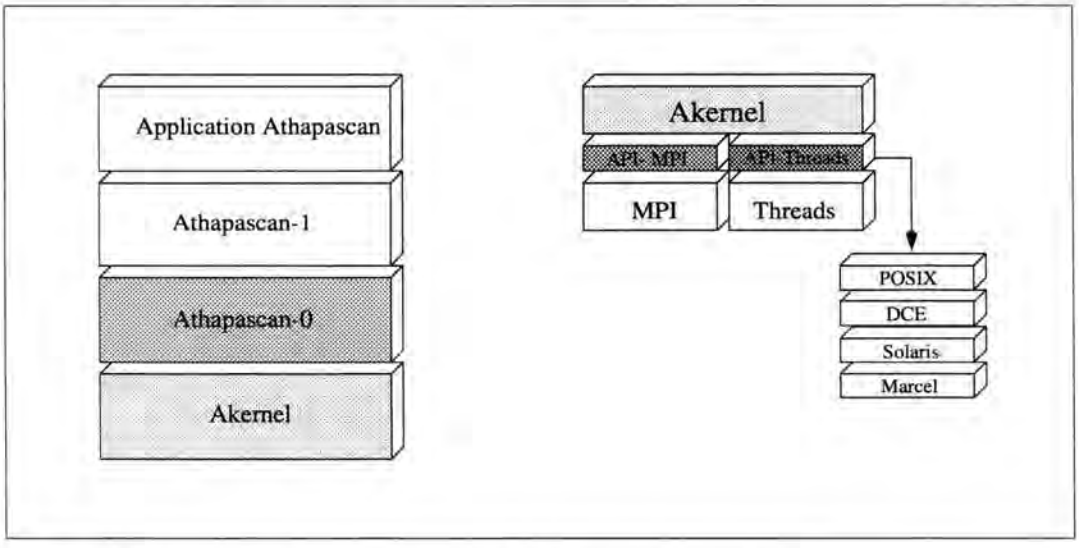

Figure 1.2 *L'architecture multi-niveau de l'environnement* ATHAPASCAN

Le niveau supérieur, **ATHAPASCAN-1** [82], est une librairie C++ basée sur le paradigme de programmation parallèle explicite par tâches concurrentes similaires à celles fournies par des langages comme Cilk [23][102] et Jade [109][136]. ATHAPASCAN-1 assure tout ce qui relève des mouvement et synchronisation de données. De plus, ATHAPASCAN-1 offre différentes stratégies d'ordonnancement pour mieux assurer la régulation de charge d'une application sur une architecture cible spécifique [33][35].

Une application ATHAPASCAN-1 est vue comme un graphe de tâches, décrit en utilisant deux mots clés : **fork** et **shared**. Les procédures qui composent un programme ATHAPASCAN-1 utilisent la primitive **fork** pour créer des unités indépen-

dantes de calcul. Un **fork** est toujours asynchrone. Dans l'abstraction ATHAPASCAN-1 il n'y a aucune communication ; tous les échanges de données sont faits de manière implicite par le biais du concept **shared** et des modes d'accès aux données (*read*, *write*, *read-write*, et *accumulate*).

La couche intermédiaire **ATHAPASCAN-0** [28][86] propose des mécanismes de création de processus légers localement et à distance accompagnés de fonctions élémentaires de communication entre eux. Le modèle de programmation offerte par ATHAPASCAN-0 est celui d'*un réseau dynamique de processus légers communicants*. Ce niveau a été initialement développé pour la programmation directe d'applications scientifiques et comme support à l'exécutif d'ATHAPASCAN-1. Néanmoins il peut être utilisé en tant que support exécutif pour d'autres langages de programmation et/ou des bibliothèques parallèles (e.g. GIVARO[84]).

Une contrainte importante du développement d'ATHAPASCAN-0 est celle de la portabilité. Ce souci a conduit à construire ATHAPASCAN-0 au dessus de deux standards : le standard de processus légers POSIX threads [100] et le standard de communication MPI [88]. Toutefois ATHAPASCAN-0 peut être facilement porté sur des bibliothèques offrant des fonctionnalités équivalentes via le niveau inférieur : **Akernel**. Ce niveau offre non seulement une machine abstraite pour ATHAPASCAN-0 mais aussi une interface uniformisée pour les processus légers et la communication. A l'état actuel, il a été porté sur différents noyaux de *threads* proches du standard POSIX (cf. figure 1.2).

1.5.2 Domaines d'applications

Les applications visées par le projet APACHE se situent dans le domaine du calcul scientifique où les algorithmes présentent un caractère irrégulier et ont un besoin de haute performance. L'intérêt d'avoir des « clients » pour l'environnement ATHAPASCAN est principalement de valider ses idées et de vérifier l'adéquation de l'approche choisie. Il est donc important de disposer d'un éventail d'applications présentant des profils distincts. Dans les sections qui suivent nous décrirons brièvement des applications réelles où ATHAPASCAN est utilisé soit directement pour l'expression du parallèlisme soit comme exécutif de base pour des couches supérieures.

Bibliothèques et méthodes mathématiques

Le calcul formel est un domaine du calcul scientifique où les algorithmes sont particulièrment gourmands en ressources de calcul et en mémoire. De plus, ces algorithmes sont très irréguliers car ils manipulent des données de taille difficilement prévisibles et non bornées. Les domaines traités sont les nombres algébriques, les polynomes et l'algèbre linéaire. L'implantation de ces algorithmes dans la bibliothèque GIVARO[84] a permis de tester sur des applications conséquentes la validité

et les performances du noyau exécutif ATHAPASCAN.

Les équations aux dérivées partielles (EDP) sont couramment employées pour analyser et prédire le comportement de certains phénomènes physiques. La solution de ces équations est faite traditionnellement par des méthodes de décompositon de domaines. Le principe est d'offrir un domaine à un processeur et de synchroniser ces processeurs aux frontières de domaines. Cependant, une zone d'intérêt particulier peut amener à un raffinement du maillage. Ce raffinement provoque des problèmes de répartition dynamique de charge de calculs et de données. Nous voulons tester l'adéquation des capacités offertes par ATHAPASCAN à résoudre ce problème. Une bibliothèque (AHPIK [38][40]) permettant ce type de calcul et de raffinement de maillage a été conçue dans le travail de thèse de Andréa Charão et doit être utilisée pour une application en océanographie et une application en turbulence aérodynamique.

Chimie, biologie et astrophysique

Le besoin de simuler le comportement de particules, comme les atomes et les molécules, est commun dans les domaines de la chimie, de la biologie et de l'astrophysique. Cette simulation suit les lois de la mécanique quantique et newtonienne où la complexité des algorithmes est égale au carré, au cube ou à la puissance 4 du nombre de particules élementaires mises en jeu. La définition de modèles plus précis pour ces phénomènes rend l'approche de modélisation et de calcul cohérent avec l'approche expérimentale. La conséquence immédiate est une augmentation de la complexité et des besoins de calcul. Le parallélisme apparaît donc comme une solution pour traiter ces modèles plus complexes.

Le projet APACHE, en coopération avec le Laboratoire de Biologie Moléculaire et Structurale du CEA-Grenoble (Y. Chapron) et avec le Laboratoire d'Astrophysique de Grenoble (P. Valiron), effectue une étude d'algorithmes et l'implantation parallèle pour des phénomènes typiques à ces domaines.

Nous citons encore l'action incitative **SimBio**[2] qui réunit comme partenaires l'INRIA Rhône-Alpes (projet APACHE), le CEA de Grenoble, le Laboratoire de Chimie Théoirque de nancy (LCTN) et l'INRIA Lorraine (projet Numath). Le but de cette action eest de concevoir une plate-forme pour la simulation moléculaire complexe. Celle-ci servira à aider la conception et le test de nouveaux algorithmes numériques parallèles ainsi qu'à servir d'outil de simulation performant.

Trafic routier

La modélisation du trafic routier peut être abordée par plusieurs types de méthodes dont la simulation à événements discrets. L'apport des techniques de modélisation et de la parallélisation des algorithmes de simulation nous permet un simulateur

[2]http ://altair.iecn.u-nancy.fr/~bernard/SIMBIO

de trafic urbain capable de rendre des services prédictifs en temps réel.

Cette application se situe dans le cadre du projet européen HIPERTRANS[129] est se fait en coopération avec B. Ycart du projet IMAG MAI, le projet SLOOP[3] et la societé Simulog.

1.5.3 Observation, évaluation et débogage

Les programmes parallèles sont en général très difficiles à déboguer du fait de l'indéterminisme d'exécution. Un autre point critique est de déterminer si la performance observée est la meilleure que l'on puisse atteindre. Il est donc désirable qu'un environnement de programmation parallèle offre à ses utilisateurs les outils nécessaires pour la mise au point des applications parallèles et l'évaluation de leurs performances. Dans le cadre du projet APACHE cet axe de recherche concerne les applications écrites en ATHAPASCAN-0 et porte, sur trois aspects : prise de traces, analyse et visualisation, et réexécution déterministe.

Traces et performance

Pour mieux comprendre et mettre au point un programme parallèle, il est souhaitable de déterminer les interactions entre les différents objets participants à une exécution, comme par example, les processeurs, les processus légers, les communications, les variables de synchronisation, etc. Dans cette optique, un traceur logiciel pour ATHAPASCAN-0 [41], assure l'enregistrement de traces lors de l'exécution d'un programme parallèle. Son principe est fondé sur l'instrumentation du code à observer.

Analyse et visualisation

Les traces obtenues lors de l'exécution d'un programme parallèle ATHAPASCAN-0 sont très difficiles à comprendre en raison du grande nombre d'événements générés par cette exécution. Pour aider la compréhension et la mise au point des programmes ATHAPASCAN-0 à partir des traces obtenues, l'environnement de visualisation Pajé [53][52] a été développé.

L'environnement Pajé propose une visualisation *post-mortem* d'un programme basée sur un diagramme espace-temps. Cette environnement possède trois caractéristiques principales : l'aptitude à représenter un nombre important d'objets ; l'aide au programmeur pour trouver interactivement les problèmes de son programme et se focaliser sur les données pertinentes ; la capacité d'extension de façon à prendre en compte l'évolution des modèles de programmation parallèle. Un travail d'intégration de Pajé à un déboguer symbolique distribué est en cours de réalisation en

[3]http ://www.inria.fr/Equipes/SLOOP-fra.html

coopération avec l'Universidade Nova de Lisboa (Portugal) et l'Universidade Federal do Rio Grande do Sul (Brésil).

Reéxecution déterministe

Un programme parallèle peut avoir un comportement correct pour une exécution et incorrecte pour une autre. Cet aspect non déterministe rend difficile l'analyse et la reproduction d'erreurs qui sont fugitives et donc particulièrement difficiles à analyser et corriger. De plus l'utilisation de moyens d'investigation tels que traceurs et débogueurs introduisent des délais d'exécution et font disparaître les erreurs qu'ils veulent capturer. La technique classique pour résoudre ce type de problème est d'enregistrer une exécution initiale et de forcer les exécutions successives à réaliser les opérations dans l'ordre enregistré. Dans ce cadre, un mécanisme original a été défini pour un sous-ensemble d'ATHAPASCAN-0 [138]. Ce travail est une action conjointe de l'équipe APACHE (J. Chassin de Kergommeaux) et de l'Université de Gent - Belgique (M. Ronsse et K. De Bosschere).

1.6 L'évolution de l'environnement ATHAPASCAN

Depuis le début du projet APACHE en 1993 l'approche choisie pour l'environnement de programmation parallèle ATHAPASCAN s'est montrée pertinente. Cependant, le noyau exécutif ATHAPASCAN-0 a dû évoluer pour accompagner les progrès scientifiques et technologiques. Dans les paragraphes qui suivent nous présenterons cette évolution jusqu'au point de la présente thèse.

La première version du noyau exécutif, appelée **ATHAPASCAN-0a**, est basée sur un modèle d'appel de procédure asynchrone. Cette version a été développée en utilisant PVM et plusieurs bibliothèques de processus légers. Son développement et son évaluation ont été faits dans le cadre de la thèse de doctorat de Michel Christaller [43].

Cette version initiale d'Athapascan reposait sur l'appel de procédure à distance où un processus léger est créé par appel de procédure. La première évolution est marquée par la généralisation de ce mécanisme pour permettre l'appel de P procédures en même temps (appel $1-P$) ou l'appel d'une procédure par N processus (appel $N-1$). Cette version, appelée **ATHAPASCAN-0**$_{mp}$ repose sur la bibliothèque de communication MPI et sur un noyau de multiprogrammation légère (POSIX). Le modèle de programmation, les choix de réalisation et l'implantation effectuées sont décrits dans [137]. Ce travail constitue la thèse de doctorat de Michel Rivière.

Parallèlement, un modèle de plus haut niveau était en cours de conception : ATHAPASCAN-1. Les besoins de mise en œuvre sur une machine parallèle IBM-SP1 [87] ont conduit à la définition et la réalisation par Ilan Ginsburg (dans le cadre

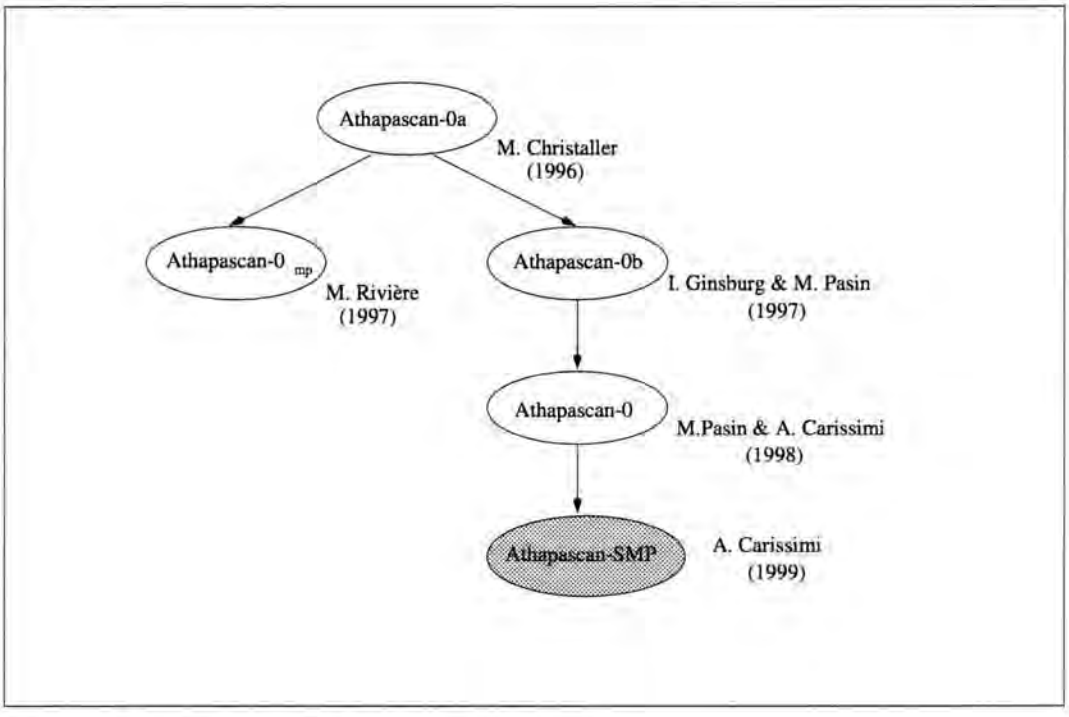

Figure 1.3 *Évolution d'Athapascan-0 (1996-1999)*

de son travail de thèse [86]) et Marcelo Pasin, d'**ATHAPASCAN-0b**. ATHAPASCAN-0b offre comme modèle de programmation l'abstraction d'un réseau de processus légers communicants. Pour cela un large éventail des primitives ont été définies. Au niveau processus légers les primitives offertes sont identiques à celles de POSIX et sont étendues par la création de processus légers à distance. Pour les communications, ATHAPASCAN-0b fournit des primitives de communications entre processus légers voisines de celles de MPI. Des opérations d'accès aux mémoires distantes (*put* et *get*) y sont ajoutées.

Ce noyau a été porté sur différentes machines parallèles (Cray T3E) ou grappes de stations et a été utilisé pour d'applications de dynamique moléculaire. Malgré une série d'ajustement l'interface de programmation est restée la même. Cette nouvelle version a été baptisée d'**ATHAPASCAN-0** et constitue le « produit » ATHAPASCAN [4] du projet APACHE.

La vulgarisation de stations multiprocesseurs entraîne à une généralisation des grappes des multiprocesseurs. Ces machines introduisent de nouveaux problèmes de mise en œuvre dûs à l'existence de plusieurs processeurs. Dans le cadre de la présente thèse, nous discuterons les choix de réalisation et l'implantation d'une nouvelle version d'Athapascan pour prendre en compte ces aspects : **ATHAPASCAN-SMP**. A nouveau, aucune altération n'a été faite sur l'interface de programmation ce qui garantit la portabilité de toutes les applications déjà réalisées.

[4] Cette version est disponible sur le WEB sur l'URL suivante : *http ://www-apache.imag.fr*

1.7 Bilan

L'emploi du parallélisme rend possible un accroissement des performances dans différents domaines d'applications. Cependant l'expression de ce parallélisme et l'exploitation des facilités mises à disposition par les architectures parallèles restent toujours difficiles à l'heure actuelle. Un programme parallèle reste complexe à élaborer car on doit coordonner les différentes entités qui coopèrent pour l'obtention de la solution. Cette coordination se présente sur différents niveaux : comment séparer un programme en tâches indépendantes ? Comment déterminer quand et par qui une tâche sera exécutée ? Comment échanger des données entre les tâches ? Pour essayer de répondre à ces questions la recherche en programmation parallèle propose différents modèles et environnements de programmation.

Une première approche est celle de la compilation. Dans ce cas-là on privilégie la parallélisation automatique et la distribution de données par un compilateur-paralléliseur. Les aspects décomposition du programme en tâches, placement, communication et synchronisation sont complètement cachés au programmeur. Cependant les efforts réalisés dans cet axe se sont montrés infructueux dans le domaine des applications irregulières. Les objectifs initiaux étaient trop ambitieux. Deux approches ont été suivies. Celles des langages fonctionnels ou logiques où le parallélisme était extrait du programme par le compilateur qui le mettait en œuvre sur une architecture cible donnée. Cette approche n'a été capable que d'exploiter le parallélisme possible dans l'interprétation du langage utilisé. Ce parallélisme ne correspond pas à celui offert par les application scientifiques mais plutôt à celui apparaissant dans des problèmes de recherche combinatoire. A contrario, la parallélisation automatique par FORTRAN a cherché à extraire le parallélisme des algorithmes séquentiels du calcul scientifique. Le résultat obtenu jusqu'a présent se réduit à la parallélisation des certaines structures du langage FORTRAN comme les boucles *for* caractéristique d'algorithmes réguliers.

La deuxième approche repose sur des modèles de programmation basés sur l'interaction entre processus. Ces modèles privilégient une découpe explicite du programme en processus et fournissent des mécanismes pour la communication. Deux paradigmes de programmation largement répandus utilisent cette approche : la programmation par échange de message et la programmation par mémoire partagée. L'implantation classique (« lourde ») de ces modèles est appropriée aux calculs réguliers. Les applications irrégulières sont caractérisées par le besoin de créer dynamiquement de nombreuses tâches de granularité très variée. La technique de multiprogrammation légère est plus appropriée à cette classe l'application.

La tendance à remplacer les ordinateurs à haute performances par des grappes de multiprocesseurs exige de manipuler deux niveaux de parallélisme : celui entre processeurs d'un même multiprocesseur et celui entre les multiprocesseurs. Le premier est efficacement exploité via la mise en œuvre d'un modèle à mémoire commune. Le second l'est via un modèle à passage de message. Un noyau pour application irrégulière doit être une implantation légère d'un modèle mixte de pro-

cessus où ceux-ci peuvent intéragir localement à un nœud (multiprocesseur) par la mémoire partagée et par des communications entre nœuds distants. Cette dualité est aujourd'hui nécessaire à l'exploitation des deux grains de parallélisme.

2

La multiprogrammation légère : le standard POSIX 1003.1

« Entre nos ennemis les plus à craindre sont souvent les plus petits »
Jean de La Fontaine, Le lion et le moucheron

Ce chapitre est consacré à la multiprogrammation légère. Initialement nous présenterons l'abstraction fondamentale sur laquelle ce concept est basé : les *threads*. Nous discuterons aussi des différentes méthodes d'implantation. L'emploi d'une nouvelle technique est justifié dans la mesure où elle rapporte des bénéfices. Nous présenterons donc les principaux points forts de la multiprogrammation légère. Ensuite nous évoquerons des styles de *threads* existants aujourd'hui, portant une attention plus particulière au standard POSIX qui représente les *threads* les plus répandus. Nous conclurons en discutant d'un aspect très important qui pose encore des problèmes : les métriques de performance et l'évaluation des *threads*.

2.1 Le modèle de processus

Le modèle de processus est ancien et remonte aux années 60 . Ce modèle définit un processus comme une exécution séquentielle d'une suite d'instructions. Un système concurrent est un ensemble de processus s'exécutant en parallèle et coopérant par la mémoire. Seules les opérations de lecture/écriture élémentaires sont atomiques. L'état que peut prendre le système correspond à n'importe quelle combinaison d'état des processus. C'est la notion d'entrelacement non déterministe. Garantir que l'état du système ne deviennent pas incorrect se ramène à synchroniser les proces-

sus entre eux de façon à interdire à certains de calculer tant que d'autres effectuent certains calculs. Les outils (sémaphores, verrous, conditions) et protocoles (producteur/consommateur, lecteur/rédacteur) sont bien connus [141][152].

Implanter un tel modèle nécessite de définir le grain de code correspondant à un processus, la stratégie d'allocation des processeurs aux processus et le grain d'entrelacement des processus entre eux. L'implantation de ce modèle repose sur le mécanisme de base dit de commutation de contexte. Il s'agit d'être capable de sauvegarder le contexte d'exécution d'un processus (pile, registres généraux, registres spécifiques d'adressage, etc) puis ultérierment de les restaurer de façon à pouvoir arrêter et ensuite redémarrer l'exécution d'un processus au gré des décisions d'ordonnancement ou des contraintes de synchronisation. C'est dans ce mécanisme que réside l'essentiel des coûts de gestion des processus.

2.1.1 La multiprogrammation lourde

C'est la technologie d'implantation la plus connue étant la plus visible pour le programmeur d'applications. Il en est d'autres plus spécifiques de la réalisation d'un système d'exploitation ou des systèmes temps réels. La principale fonction d'un système d'exploitation est de fournir un environnement d'exécution pour les applications (programmes) de ses utilisateurs. Le processus correspond alors à l'exécution du programme séquentiel d'un utilisateur [152]. Le grain de code est celui du fichier exécutable. Le coût de création est celui du chargement d'un fichier en mémoire.

Dans des systèmes d'exploitation basés sur UNIX, ou NT, plusieurs processus (programmes) existent au même instant. Le rôle du système est de gérer au mieux les ressources disponibles (processus, mémoire) mais aussi de gérer de façon équitable leur allocation aux processus. En général, cette gestion repose sur un recouvrement calcul-entrée/sortie et une attribution équitable des processeurs aux processus. Le premier point est assuré par une suspension du processus effectuant une requête d'entrée/sortie. Le processeur est alors attribué à un autre processus. Le processus en attente d'E/S redevient candidat à un processeur en fin d'E/S. Le second point essaie d'attribuer sur une période donnée la même pourcentage de temps d'accès au processeur. Cette attribution est évaluée sur fin d'E/S ou périodiquement (*quantum* ou tranche de temps).

Cette technique d'implantation est dite « lourde » car en plus de la lourdeur de création, l'opération de commutation de contexte est elle-même lourde. En effet la définition d'un processus comme l'exécution du programme d'un utilisateur exige la mise en place de mécanismes de protection isolant les espaces d'adressage des processus (mémoire virtuelle) ainsi que des mécanismes de contrôle d'accès aux fichiers, des mécanismes de facturation, etc. La mise en place de ces mécanismes se traduit par des opérations de sauvegarde et restauration de registres spécialisés des processeurs (registres d'espace virtuel ou de segments, registres de taille de pages,

vidage de cache, etc). Le coût de commutation de contexte s'élève en proportion.

Cet implantation où tous les processus sont indépendants présentent de sérieuses limitations en particulier lorsque les processus doivent interagir. Par exemple, dans une application de type client-serveur, le serveur doit sérialiser les traitements demandés par plusieurs de ses clients, ou sinon, créer des processus esclaves pour les traiter. Or comme chaque processus esclave a son propre espace d'adressage il n'est pas possible de partager directement des données entre esclaves et le serveur. Pour résoudre ces problèmes, des mécanismes de synchronisation (sémaphores, system V), de définition de zones de mémoire partagées (*shared memory system* V ou Berkeley) et de communication (*pipe, message queue, etc...*) ont été ajoutés aux systèmes d'exploitation. Cependant, l'implantation reste lourde car le grain de création de processus reste important (chargement de fichier) ainsi que le coût de commutation dans la mesure où la protection doit être assurée. Le besoin croissant d'efficacité des applications « serveurs internet », des applications parallèles a nécessité un allégement de ces coûts : c'est la multiprogrammation légère.

2.1.2 La multiprogrammation légère

La multiprogrammation légère est essentiellement une implantation à grain fin du concept de processus. Elle peut être mise en place par des logiciels ou d'une façon matérielle comme pour la machine Tera [6] ou la machine Eart-Mana[97], par exemple. Le terme unité pour une implantation légère de processus est celui de ***thread*** (processus légers). Sa légèréte provient du :

- grain de code utilisé. C'est la procédure pour les implantations logicielles. C'est l'expression pour une implantation matérielle ;
- coût de commutation de contexte. Il se réduit à la sauvegarde/restauration des registres nécessaires à l'arrêt/reprise d'exécution d'une procédure.

Les *threads* peuvent coopérer via des données communes et des opérateurs de synchronisation classiques. Un « programme parallèle léger » est alors constitué des codes et données statique, d'un tas, des piles et contextes des processus. A ce dernier point prêt, l'organisation de la mémoire d'un programme parallèle léger est idéntique à celle d'un programme séquentiel. Cette propriété sera exploitée par les techniques d'implantation de *threads*.

Un programme parallèle léger est une entité composée de deux parties : un ensemble de *threads* et une collection de ressources [161]. Un *thread* est un objet dynamique qui est en soi un point de contrôle d'un processus et est responsable de l'exécution d'une procédure. Les ressources, soit l'espace d'adressage, les permissions d'accès, les fichiers, les quotas, etc, sont partagés par les *threads*. On peut voir un processus lourd d'UNIX comme l'exécution d'un programme parallèle par un seul *thread*. Les différentes implantations du concept de *thread* vont toutes proposer une même représentation des ressources que le programme executé soit séquentiel

ou parallèle. Elles différeront par la façon d'exécuter les *threads* c'est à dire la méthode d'allocation des processeurs aux *threads*.

2.2 L'implantation de *threads*

Le rôle du système d'exploitation reste identique. Il doit bien gérer les ressources et assurer leur partage équitable entre les différents programmes utilisateurs. Ceci veut dire qu'il doit pouvoir gérer plusieurs programmes parallèles d'utilisateurs différents. Deux façons extrêmes peuvent être utilisées pour implanter des *threads*. Dans la première, le système d'exploitation est inchangé et exécute un programme utilisateur parmi d'autres. Le concept de *thread* est mis en œuvre au sein du programme utilisateur par une bibliothèque. Dans la seconde, le système d'exploitation gère les *threads* d'un programme parallèle. Une dernière méthode essaie de combiner les avantages des deux premières.

2.2.1 Le modèle N :1

La première méthode pour l'implantation de *threads* est connue comme **Many-to-One**, ou simplement, N :1. Le système crée un processus lourd pour exécuter un programme parallèle de façon identique à l'exécution d'un programme séquentiel. Ce processus lourd est activé et désactivé au gré des opérations d'entrée/sortie ou des décisions d'ordonnancement du système. Le concept de *thread* est implanté par une bibliothèque permettant de créer un *thread* (allocation de pile) de l'activer et le désactiver sur opération de synchronisation ou interruption d'horloge. Cette implantation est appelée aussi *user level threads* car elle ne fait pas intervenir le système d'exploitation.

Le noyau exécutif (*run time system*) considère le processus lourd comme un processeur virtuel et réalise le contrôle l'exécution des *threads* : ordonnancement, synchronisation, etc. Le principal avantage de ce modèle est sa souplesse. Comme toutes les opération de contrôle des *threads* sont faites dans l'espace utilisateur, aucune interaction entre les *threads* ne nécessite d'appels systèmes. Le temps de commutation de contexte entre *threads* est réduit au minimum nécessaire. Ce modèle est représente de façon schématisé par la figure 2.1. La bibliothèque assure le partage du processus lourd entre les *threads*. Le système partage le(s) processeurs(s) entre les processus lourds.

Le grand désavantage de ce modèle est justement le fait que les *threads* sont simulés sur un seul flot d'exécution réel, celui du processus lourd les « portant ». Ceci implique que le parallélisme réel d'un SMP ne peut être exploité par un programme parallèle. Il ne peut servir qu'à exécuter plusieurs programmes parallèle en même temps. Un autre problème provient de la multiprogrammation système : une entrée/sortie faite à l'initiative d'un *thread* bloque le processus lourd portant

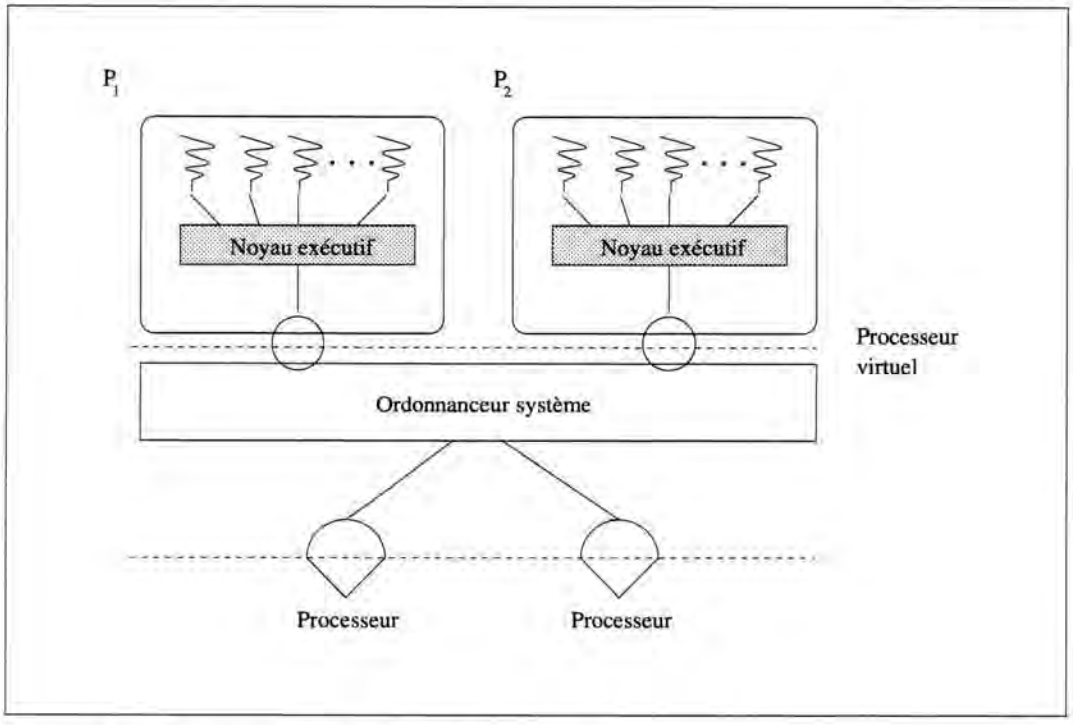

Figure 2.1 *Le modèle N :1* - threads *niveau utilisateur*

le *thread*. Les *threads* ne peuvent servir directement au recouvrement calcul-communication. Il sera nécessaire de contourner ces limitation dans l'intégration multi-programmation légère et communication. Nous reviendrons à ce point au chapitre 4.

2.2.2 Le modèle 1 :1

Le modèle **One-to-One** (1 :1) résout les deux problèmes mentionnés plus haut : exploitation du parallélisme SMP et le recouvrement calcul-entrée/sortie. Pour le faire le système d'exploitation doit être conçu pour gérer explicitement les *threads* ou plus exactement des processeurs virtuels partageant une même mémoire virtuelle. [1].

Dans un modèle 1 :1, chaque *thread* d'un programme utilisateur est attaché à un processeur virtuel propre. Les *threads* utilisateurs sont visibles du système d'exploitation qui les gère. L'avantage principal de cette méthode est le fait qu'elle permet aux *threads* d'un même processus de s'exécuter simultanément sur des machines SMP. De plus, si le *thread* d'un programme est bloqué sur un appel système, les autres *threads* de ce programme (même sur un monoprocesseur) peuvent continuer à s'exécuter. La figure 2.2 illustre les principes de ce modèle.

[1] Sur différents systèmes on retrouve les dénominations lightweight process (Solaris), sproc (Irix), virtual processor (AIX), etc...

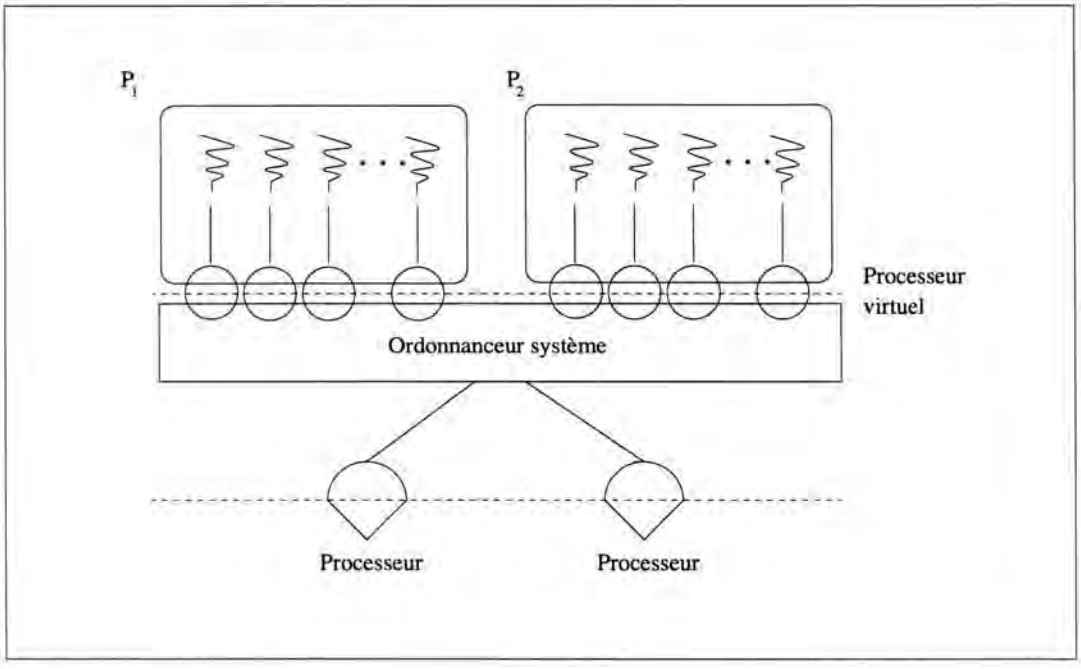

Figure 2.2 *Le modèle 1 :1* - threads *niveau système*

Le modèle 1 :1 présente certains inconvénients. La création et la gestion d'un « processeur virtuel » c'est à dire des entités « porteuses » des *threads*, par le noyau, sont des opérations plus coûteuses si on les compare à celles du modèle N :1. En effet, généralement, on doit commuter de *thread* d'un programme à un *thread* d'un autre programme c'est à dire payer le prix d'une commutation « lourde ». De plus l'ordonnancement des *threads* est fait, à présent, par le noyau système et l'utilisateur n'a aucune, ou peu, d'influence sur lui. Un autre désavantage apparaît par rapport à la consommation des ressources systèmes. Le nombre des « processeurs virtuels » est normalement une ressource système limitée donc le nombre maximal de *threads* pour tout le système aussi. Théoriquement, sur un modèle N :1 nous pouvons avoir autant de *threads* que l'on souhaite[2].

2.2.3 Le modèle M :N

Black [22] distingue pour un algorithme parallèle la **concurrence** du **parallélisme**. Le dégré de concurrence est le nombre d'exécutions parallèles qu'il est possible d'avoir pour un algorithme en supposant un nombre infini de processeurs. Dans notre cas c'est grossièrement le nombre de *threads* d'un programme. Le parallélisme est le nombre d'exécution parallèles qu'il est physiquement possible d'avoir. Si l'on ignore les problèmes de multiprogrammation sur entrée/sortie c'est le *minimum* {nombre de processeurs, degré de concurrence}.

Chacun des deux modèles, N :1 et 1 :1, présentent leurs avantages et incon-

[2]La limitation est en fait par rapport à la capacité de la mémoire virtuelle.

vénients. Le modèle 1 :1 permet l'exploitation de la multiprogrammation et du parallélisme SMP. Cependant les *threads* de ce modèle sont gourmands en ressources systèmes ce qui est gênant pour le développement d'applications concurrentes à grand nombre de *threads* (degré concurrence maximum). Le modèle N :1 n'a pas cette limitation mais a un de degré de parallélisme limité à 1.

Le modèle M :N, ou **Many-to-Many**, est une combinaison des deux modèles précédents en vue de préserver leurs avantages et éliminer leurs inconvénients. Sur un modèle de ce type, le noyau système est capable d'offrir plusieurs entités « porteuses » de *threads* ou « processeurs virtuels ». L'attribution de ceux-ci aux *threads* est faite par une librarie au niveau utilisateur (fig. 2.3).

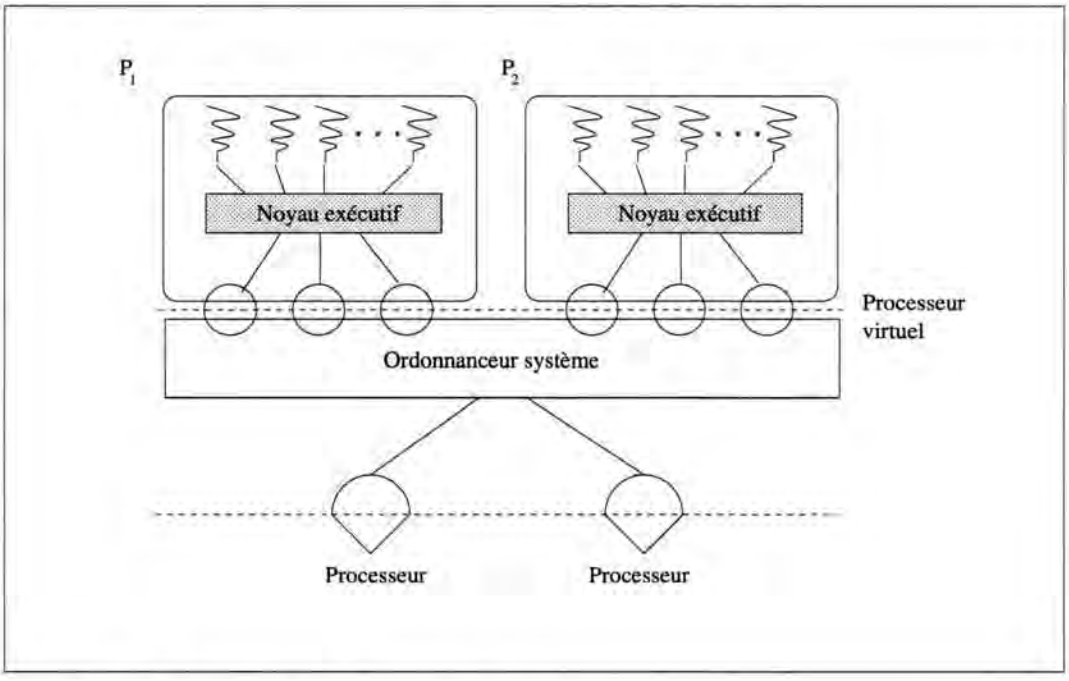

Figure 2.3 *Le modèle M :N* - threads *à deux niveaux*

Le modèle M :N présente un ordonnancement à deux niveaux : la librairie au niveau utilisateur détermine quels *threads* doivent s'exécuter sur quels processeurs du « multiprocesseur virtuel » simulé par le noyau système. Ceux-ci, à leur tour, sont ordonnancés par le noyau du système d'exploitation sur les processeurs physiques disponibles. Les *threads* sont créés et gérés par la librairie donc de façon peu coûteuse. L'utilisation pour leur exécution de plusieurs processeurs virtuels permet d'exploiter le parallélisme SMP et d'assurer le recouvrement calcul-entrée/sortie.

Ce modèle d'implantation de *threads* a pour origine l'article « *scheduler activations* » proposé par Anderson[9]. Le système Solaris de Sun offre une telle fonctionnalité[63][131].

2.3 Les styles de *threads*

Aujourd'hui les styles de *threads* qui dominent la technologie informatique sont les **POSIX *threads***, les *threads* de **Microsoft** et les *threads* de **Java** [112]. Bien que ces styles différent par les implantations et les interfaces de programmation (API), les concepts fondamentaux restent essentiellement les mêmes.

Le style POSIX a vu son interface et sa sémantique définie par le standard IEEE POSIX 1003.1[100]. Ce standard est largement accepté aujourd'hui et la plupart des constructeurs en proposent une implantation sur leurs systèmes UNIX. Le standard voisin **UNIX international threads (UI)**[3] est peu utilisé.

D'autre part, nous trouvons des bibliothèques propriétaires du style **Microsoft**. La bibliothèque *Win32 threads*[135] utilisée dans les systèmes d'exploitation windows95/98 et windows NT, et la bibliothèque *OS/2 threads* développée à l'origine par Microsoft pour le système d'exploitation OS/2 (IBM) en font partie.

Les styles POSIX *threads* et Win32 *threads* se présentent comme des interfaces applicatifs C/C++. Le style Java est en revanche totalement intégré au langage de programmation objet **Java**[14]. En Java le concept de thread constitue une classe à partir de laquelle nous pouvons instancier autant de *threads* que l'on veut. Le concept de *thread* est intégré dans la *Java Virtual Machine* (JVM). Aucune sémantique précise n'a été definie et les implantations peuvent produire des comportements différents d'un même programme.

2.4 Les *threads* POSIX

Le **standard POSIX ANSI/IEEE 1003.1**[100] (aussi appelé ISO/IEC 9945-1 : 1996) définit le comportement et l'interface de bibliothèques de *threads*. Les bibliothèques qui suivent ce standard sont aussi appelées *Pthreads* ou encore POSIX.1c *threads* puisque les *threads* sont traités à la section 1003.1c de ce standard.

La norme POSIX donne une définition detaillée des *threads* et des proprietés qu'une implantation doit vérifier. La plupart des implantations se conforment à un sous ensemble du standard. Dans le cadre du présent travail nous sommes particulièrement intéressés par la portabilité. En outre, les fonctionnalités offertes sont assez semblables à celles d'autres noyaux. C'est pour cela que nous avons choisi d'utiliser les *threads* POSIX que nous décrirons par la suite.

[3]Le constructeur Sun (SunSoft) a utilisé ce standard lors du développement de sa bibliothèque Solaris *threads*. Les deux standards sont assez proches, au point qu'à partir de la version Solaris 2.5 il est possible d'utiliser les deux standards au sein d'un programme. En effet, l'interface POSIX disponible sur Solaris est contruite sur les Solaris *threads*.

2.4.1 Le cycle de vie d'un *thread* POSIX

Créer un *thread* signifie le caractériser par des **attributs**, lui associer une procédure à exécuter, et éventuellement des paramètres. L'*attribut* est un objet qui définit les caractéristiques du *thread* créé. Les attributs spécifiés par la norme POSIX définissent des aspects liés à l'interaction avec les autres *threads* et le système d'exploitation, à l'ordonnancement et à la terminaison. Ces attributs sont :

- **detachstate :** contrôle comment la librairie de *threads* doit procéder à la terminaison d'un *thread*. Deux modes possibles sont : **joinable** et **detached**. Un *thread* en mode *joinable* exige qu'un autre *thread* se synchronise avec sa fin et de récupère le résultat. Les ressources mémoires occupées par un *thread joinable* ne sont libérées complètement que lorsqu'un autre *thread* réalise la synchronisation de fin par le biais de la primitive *join*. Au contraire, un *thread detached* libère complètement les ressources allouées lors de sa terminaison. Se synchroniser par un *join* est alors impossible.
- **schedpolicy** et **schedparam :** définissent la politique d'ordonnancement et ses paramètres (e.g., priorités) pour le *thread* créé. Additionellement, on trouve l'attribut **inheritsched** qui associe au *thread* créé la politique et les paramètres d'ordonnancement du *thread* créateur.
- **scope :** détermine le domaine d'ordonnancement du *thread* : **system** ou **process**. Un *thread* qui a une portée *system* se présente à l'ordonnanceur comme une unité independante (modèle 1 : 1). La portée *process* signifie que le *thread* est ordonnancé au sein d'un processus (modèle N : 1). On notera que le standard POSIX ignore les implantations de type M :N.

Après que ses attributs aient été définis un *thread* est créé par un appel à la primitive *pthread_create()*. Un identificateur unique par le *thread* est retourné par cet appel. Cette primitive accepte le passage de paramètres pour la procédure associée au *thread* par le biais d'un pointeur de type *void*. Si l'on souhaite plus d'une valeur comme paramètre, il faut définir une structure et passer son adresse en paramètre. Le retour, pour les *threads* d'attribut *joinable*, est fait de façon similaire. Le cycle de vie d'un *thread* POSIX est représenté par le diagramme de la figure 2.4.

Initialement quand un *thread* est créé, il est mis immédiatement dans un état *Runnable* à partir duquel il évolue jusqu'à sa terminaison. Lorsque l'ordonnanceur du système le choisit pour exécution, le *thread* passe à l'état *Active*. Pendant l'exécution, le *thread* peut éventuellement réaliser des opérations de synchronisation comme l'utilisation de verrous, de sémaphores, dépendre d'un événement externe (variables de conditions), ou encore réaliser une opération considérée comme lente (entrée sortie, par exemple). Dans ces situations, le *thread* rentre dans un état *Blocked* où il attendra l'occurrence des événements devant le débloquer. Il est possible aussi que pendant l'exécution d'un *thread*, un autre *thread* plus prioritaire redevienne prêt (*Runnable*). Dans ce cas une préemption a lieu. Le *thread* actif est arrêté dans un état prêt. Le *thread* prioritaire devient actif. Les états *Destroyed* et *Zombie* sont asso-

ciés à la terminaison d'un *thread*. L'état *Destroyed* indique seulement que le *thread* n'existe plus. L'état *Zombie* est un état intermédiaire où le *thread* attend qu'un autre *thread* execute le *join*.

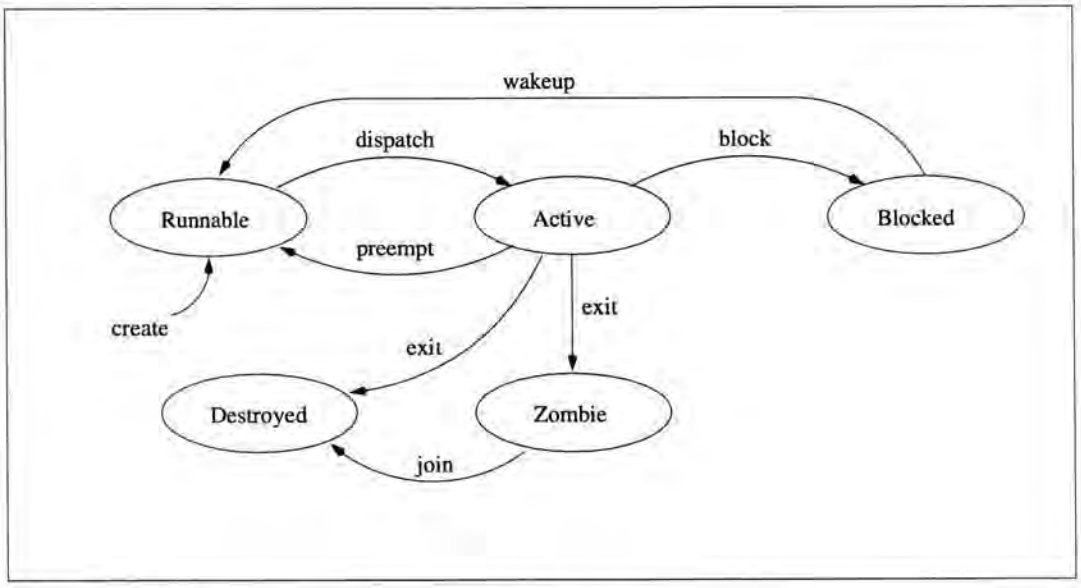

Figure 2.4 *Le diagrame d'état des* POSIX *threads*

La terminaison d'un *thread* est faite explicitement par un appel à la primitive *pthread_exit()* ou implicitement par la fin de la procédure qu'il exécute. Toutefois il est parfois nécessaire de forcer la terminaison d'un *thread* depuis un autre *thread* c'est le concept d' **annulation** (*cancellation*).

L'**annulation** (*pthread_cancel()*) permet à un *thread* de demander la terminaison d'un autre *thread* : plus précisément, le *thread* qui demande le *cancel* le signale au *thread* cible. Celui ci en fonction de son état peut ignorer cette demande, l'exécuter immédiatement (*asynchronous cancellation*), ou la différer (*deferred cancellation*) jusqu'au moment où son état le permet. Les états d'exécution où la terminaison forcée est possible sont appelés points d'annulation (*cancelation points*). Certains sont définis dans la norme et sont en général des opérateurs de la bibliothèque POSIX et des opérateurs systèmes impliquant des blocages.

Le contrôle de la prise en compte de l'annulation est nécessaire pour éviter de laisser des données dans un état incohérent. Deux outils sont donnés pour traiter ce problème : (a) des opérateurs valider/invalider/différer un ordre d'annulation ; et (b) un mécanisme d'exécution de procédures de nettoyage au moment de l'annulation (*clean up handler*).

2.4.2 Priorités et politiques d'ordonnancement

L'exécution d'un *thread* est caracterisée par une séquence de transitions entre les états *Runnable*, *Active* et *Blocked*. Les transitions entre états *Active* $\leftrightarrow$ *Runnable*

et entre les états *Blocked* $\leftrightarrow$ *Runnable* se font en fonction des priorités et de la politique d'ordonnancement utilisées par le système.

Un ordonnanceur peut être **préemptif** ou **non-préemptif** indépendamment de la politique d'ordonnancement employée. Dans un ordonnanceur non-préemptif, le *thread* s'exécute jusqu'à la fin, jusqu'à ce qu'une opération le bloque ou, qu'il « passe la main » à un autre *thread* (appel du type *yield()*). Dans un ordonnaceur non-préemptif, un *thread* ne se desactive qu'en des points précis du programme. Au contraire dans un ordonnanceur préemptif, un *thread* peut être désactivé par l'ordonnanceur en n'importe quel point du programme[4]. Les ordonnanceurs POSIX sont tous préemptifs du fait des priorités.

La gestion de *threads* se fait par priorité. Son implantation est basée sur les listes de *threads* prêts à s'exécuter (*Runnable*) organisées en fonction de leur priorité. L'ordonnanceur choisira donc le premier *thread* de la liste de priorités le plus élevé (non vide, bien sûr). La priorité d'un *thread* est définie par une paire {politique d'ordonanncement, priorité relative à la politique}. Cette dernière est un simple entier. Une politique d'ordonnancement précise les régles d'applications des priorités dans la politique considérée. Les politiques sont elles même ordonnées par priorité. Le standard POSIX prévoit les trois politiques suivantes dans l'ordre de priorité croissante :

SCHED_FIFO (First In, First Out) : Un ordonnanceur de ce type gère les *threads* selon les règles suivantes :

- quand un *thread* est préempté, il est remis en tête de sa liste de priorité ;
- quand un *thread* doit être débloqué (*Blocked* $\rightarrow$ *Runnable*), on choisit le plus prioritaire des bloqués candidats au réveil. Ceci se produit sur les opérations de synchronisation (relâche de verrou, de signalisation d'une variable de condition). Le *thread* est mis en queue de sa file de priorité.
- si le *thread* actif change sa priorité, il est mis à la fin de la liste correspondant à sa nouvelle priorité s'il doit être préempté (baisse de priorité)
- quand le *thread* fait un appel de type *yield()*, il est mis à la fin de sa liste de sa priorité ;

Cette politique est dit FIFO car elle réalise les transitions quittant un état à l'autre dans l'ordre d'arrivée à ce nouvel état pour une priorité donnée.

SCHED_RR (Round-Robin) : Un ordonnanceur Round-Robin[5] agit de façon similaire. De plus un *thread* actif peut avoir son exécution interrompue au bout d'une période de temps prédéfinie (*quantum*). Il est remis en queue de sa liste de priorité. Cette politique assure un avancement équitable des *threads* prêts pour une priorité donnée.

[4] Cela dépend complètement de l'implantation.

[5] tourniquet

SCHED_OTHER : La politique d'ordonnancement dépend de l'implantation. C'est généralement un ordonnancement à la UNIX qui essaie d'attribuer à tous les processus (*threads*) une durée d'exécution équivalente. Ceci est fait par un calcul de priorité dynamique qui fait qu'un *thread* longtemps bloqué « gagne » le droit d'utiliser plus de temps processeur pour « rattraper » les autres processus. Ce mécanisme évite la famine et à long terme fait que le temps d'exécution est divisé de façon équitable parmi les *threads*.

La partie de la norme traitant de l'ordonnancement est dite optionnelle. La seule contrainte est celle des priorités entre politiques d'ordonnancement. La conséquence est que la plupart des implantations offre simplement la dernière politique[6].

2.4.3 Les *threads* et leur espace d'adressage

Par définition, les *threads* partagent un même espace d'adressage[7]. Les *threads* peuvent communiquer entre eux par partage des données communes. Classiquement, des procédures peuvent échanger des données par les variables globales, les variables dynamiques du tas et les variables de la pile. Dans ce dernier cas, on doit prendre en compte la durée de vie fugace d'un variable locale.

À la création, une **pile** est associée à chaque *thread*. Cette pile sert au mécanisme de passage de paramètres ainsi que à la gestion des appels aux fonctions réalisés par le *thread*. Les variables locales, qui ne sont visibles que dans le contexte du *thread* sont allouées sur cette pile, donc accessibles exclusivement par le *thread*. Il apparait un problème avec les variables globales. Dans un langage séquentiel, celles-ci jouent un double rôle : (a) permettre le partage des variable entre différentes procédures ; et (b) préserver une valeur entre appels successifs d'une même série d'appels.

Dans un contexte *multithread*, les variables globales sont communes à tous les *threads*. Le premier rôle reste assuré. En revanche une variable globale ne peut plus préserver une valeur au cours d'une série d'appels issus d'un même *thread*. La valeur contenue par la variable est la dernière valeur attribuée par un *thread*. Un exemple classique est celui de la variable globale *« errno »*. Un *thread* réalise un appel système et génère une erreur. Avant qu'il puisse vérifier la valeur de la variable *« errno »*, un autre *thread* exécute un nouvel appel système et change sa valeur : c'est un effet de bord. La **zone spécifique aux données privées** (*thread-specific data*, ou TSD) est destinée à contenir des variables globales à un *thread*.

Les variables globales privées sont identifiées par des clefs. Les clefs sont uniques et globales à tous les *threads*. Elles permettent de définir entre les procédures une identification conventionnelle des variables globales. Chaque *thread* pourra alors

[6]Dans le cas des noyaux POSIX temps réel, ce sont les deux premières qui sont implantées.
[7]Celui du processus lourd initialement créé pour le premier *thread*.

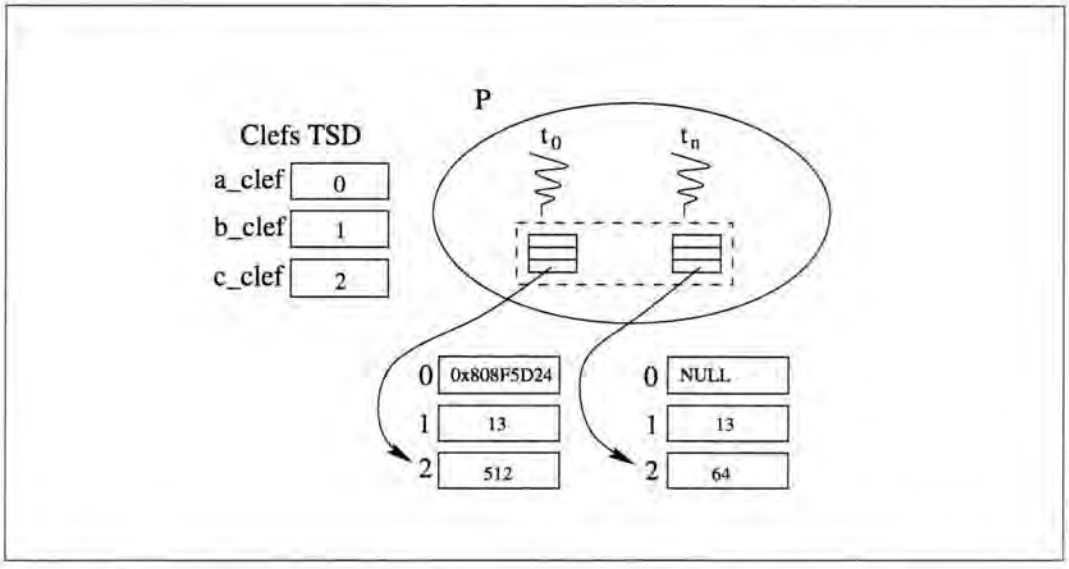

Figure 2.5 *La zone spécifique aux données privées (*thread-specificdata, *ou TSD)*

affecter une valeur à son instance privée de cette variable lors des appels aux procédures l'utilisant. L'implantation est généralement faite par un tableau de variables par *thread*. La gestion des clef est assuré par un compteur global incrementé de façon atomique pour génerer des clef uniques (figure 2.5).

2.4.4 Les primitives de synchronisation

Le partage de données entre *threads* pose le problème du maintien de cohérence de ces données. Le maintien de cohérence des données partagées repose sur deux capacités :

- la première consiste à assurer l'atomicité des opérateurs de consultation et modification. Ce problème provient du fait qu'une variable ne peut être lue ou écrite en une seule opération matérielle. Ainsi, une valeur lue par un *thread* pourrait être une combinaison arbitraire des valeurs élémentaires écrites par d'autres *threads*.
- la seconde consiste à arrêter l'exécution des processus lorsque des variables ne vérifient pas une certaine proprieté.

Des nombreux concepts de programmation ont été introduits pour traiter ce problème (sémaphore, moniteur, etc). POSIX se ratache à la famille des « section critiques conditionnelle » [94], où un processus peut de façon atomique tester une conditon et effectuer une action si elle est vraie puis signaler que d'autres conditions sont devenues vraies.

Le standard POSIX définit les deux concepts de base permettant la programmation des sections critiques conditionnelles : les **verrous** (*mutex*) et les **variables de**

condition (*condition variables*). Le premier concept permet d'assurer la cohérence d'une suite d'actions par exclusion mutuelle d'accès à une séquence de code (section critique). Le second concept est destinée à permettre les attentes et signalisation de conditions.

Au dessus de ces deux primitives, il est possible de construire des mécanismes classiques de synchronisation comme des moniteurs, des sémaphores, et des verrous de type *read/write lock.*

Les verrous

Le **verrou** est la plus simple primitive de synchronisation basée sur deux états : verrouillé et non verrouillé. Une section critique est donc une partie de code parenthésé par une opération de verrouillage et une opération de déverrouillage. Le premier *thread* qui veut exécuter une section critique prend le verrou de protection *pthread_mutex_lock()*, ce qui aura par conséquence de bloquer tout autre *thread* essayant de rentrer dans cette même section critique. Quand le *thread* qui a réussi à prendre le verrou termine l'exécution de la section critique il la libère (fig. 2.6.a, *pthread_mutex_unlock()*). Un des *thread* en attente de ce verrou sera mis dans l'état *Runnable*. Le *thread* débloqué sera celui de la plus haute priorité ce qui pose des problèmes spécifiques (voir aussi l'item « Le problème d'inversion de priorité »).

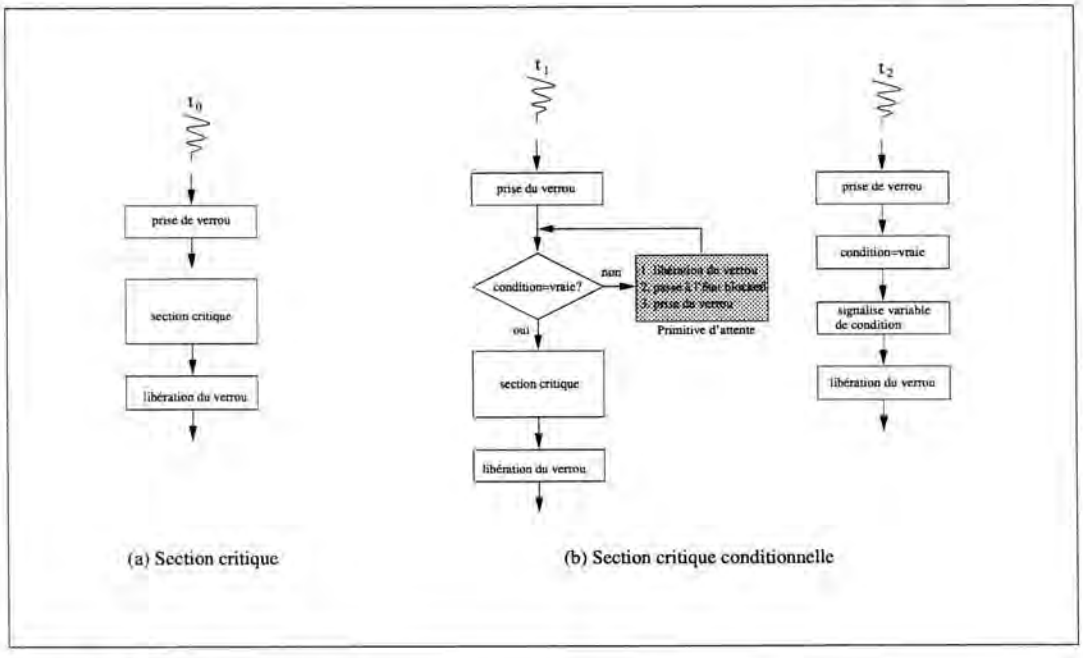

Figure 2.6 *Les primitives de synchronisation*

Variable de condition

Une autre situation de synchronisation est celle où un *thread* a besoin d'attendre qu'une condition spécifique soit satisfaite pour continuer son exécution. Cette fonction est donnée par les **variables de condition** que l'on peut signaler et attendre. Leur sémantique de fonctionnement impose l'utilisation conjointe d'un verrou à la variable de condition.

La primitive *pthread_cond_wait()* provoque le blocage du *thread* qui la réalise jusqu'à ce que la condition soit signalée. Son exécution libère le verrou associé à la condition avant de bloquer le *thread* appelant. Lorsque la condition est signalée via *pthread_cond_signal()* ou *pthread_cond_broadcast()*. L'action de blocage et libération du verrou est implantée de façon atomique. La primitive *pthread_cond_signal()* débloque au moins un *thread* bloqué sur la condition, *pthread_cond_broadcast()* débloque tous. La figure 2.6.b illustre ce mécanisme.

Le problème d'inversion de priorités

Quand nous sommes en présence d'ordonnanceurs à priorité, il peut se produire un phénomène connu sous le nom d'**inversion de priorités**. Un *thread* de plus basse priorité empêche l'exécution d'un *thread* plus prioritaire. Deux situations peuvent provoquer l'apparition de ce phénomène (fig. 2.7).

Dans la première situation, un *thread* de basse priorité possède un verrou qu'un *thread* de plus haute priorité essaie d'acquérir. Un *thread* plus prioritaire doit attendre la libération du verrou par le *thread* de plus basse priorité. Cette situation est appelée **inversion bornée de priorités** (*bounded priority*) car la durée de l'attente est celle de la section critique. Elle est normale et inévitable si les deux *threads* partagent une donnée. La deuxième situation surgit quand d'autres *threads* de priorité intermédiaire empêchent l'exécution du *thread* de plus basse priorité, par conséquent, ils retardent la libération du verrou attendu par le *thread* de plus haute priorité. Cette situation dure tant que ces *threads* sont exécutables d'où son nom : **inversion non bornée de priorités** (*unbounded priority inversion*).

Le standard POSIX offre deux solutions au problème de l'inversion non bornée en permettant d'associer à un verrou une des deux régles de changement de priorités : **priorité plafonnée** (*priority ceiling*) et **héritage de priorité** (*priority inheritance*).

La première, la priorité plafonnée, consiste à augmenter la priorité du *thread* prenant un verrou à une valeur pré-déterminée. Pour éviter le phénomène d'inversion non bornée, cette valeur doit être au moins égale[8] à la priorité la plus élevée des *threads* qui seront en compétition sur le verrou. Quand le verrou est libéré, le *thread* revient à sa priorité initiale. Cette méthode est efficace et prévient bien l'inversion. Elle a cependant deux inconvénients. Il faut connaître tous les *threads*

[8]Si elle est immédiatement supérieure, on évite toute inversion.

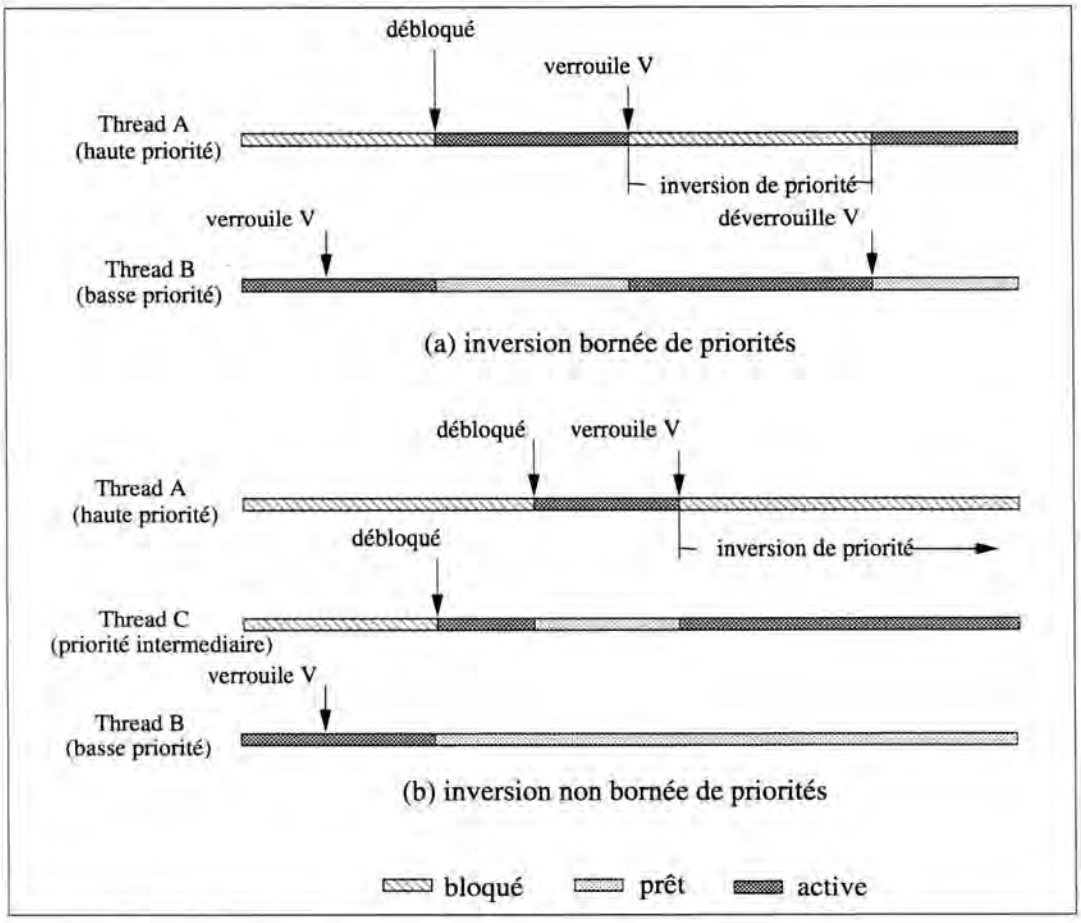

Figure 2.7 *Le problème d'inversion de priorités*

pouvant accéder à un verrou pour déterminer le plafond. Par ailleurs, cette solution peut interdire à un *thread* de priorité intermédiaire de tourner même si aucun *thread* de priorité élevée ne demande le verrou (c'est à dire en l'absence de risques d'inversion).

La régle d'héritage de priorité corrige cet inconvenient en conservant la priorité du *thread* qui détient un verrou jusqu'au moment où d'autres *threads* veulent l'acquérir. A cet instant le *thread* voit sa priorité augmenter au niveau de celle du *thread* le plus prioritaire demandent de ce verrou. En d'autre termes, le *thread* qui possède le verrou « hérite » de la priorité du *thread* bloqué le plus prioritaire. De cette façon, le *thread*, durant la période où il détient le verrou, ne pourra plus être préempté par d'autres *threads* moins prioritaires que ceux qui attendent le verrou. Le changement de priorité n'est effectif que lors de l'apparition du risque d'inversion. Cette souplesse se paie par des commutations supplémentaires. Par exemple, si l'on reprend la situation initiale d'inversion, le *thread* moins prioritaire est préempté quand le *thread* prioritaire devient actif. Il ne voit sa priorite augmentée, et ne redevient actif, qu'au moment où le *thread* prioritaire tente d'acquerir le verrou.

Le choix de la politique à utiliser pour traiter le problème d'inversion de prior-

ité est laissé à charge du programmeur. Le comportement du verrou est fixé à son initialisation par un attribut.

2.4.5 Les *Threads* POSIX et le système UNIX

Tableau 2.1 *Les* threads *POSIX et le système UNIX*

Notes :
1. Single UNIX Specification, version 2, (UNIX98) qui inclut POSIX 1003.1c-1995.
2. Implante partiellement SUSv2.
3. Il est nécessaire d'installer des *patchs* système.
4. Dépend de la bibliothèque installée.
5. SCS fait référence à System Contention Scope (ordonnancement au niveau système). PCS fait référence à Process Contention Scope (ordonnancement à deux niveaux, système et processus).
6. La bibliothèque CMA est une implantation de threads selon le modèle N :1. Elle est partie intégrante de DCE.

Fabricant	Système	Modèle	interface	Ordonnanceur[5]
Sun	Solaris 2.7	M :N	SUSv2[1], Solaris	SCS, PCS
	Solaris 2.6	M :N	POSIX 1003.1c, Solaris	SCS, PCS
	Solaris 2.5	M :N	POSIX 1003.1c, Solaris	SCS, PCS
Compaq	Tru64 UNIX 5.0	M :N	POSIX 1003.1c[2], DCE, CMA[6]	SCS, PCS
	Digital UNIX 4.0	M :N	POSIX 1003.1c, DCE, CMA[6]	SCS, PCS
	DEC OSF/1 3.2	1 :1	DCE threads, CMA[6]	SCS
HP	HP-UX 11.0	1 :1	SUSv2[1]	SCS
	HP-UX 10.30	1 :1	POSIX 1003.1c	SCS
	HP-UX 10.20	N :1	DEC threads	PCS
IBM	AIX 4.3.1	M :N	SUSv2[1]	SCS,PCS
	AIX 4.3	1 :1	POSIX 1003.1c[2]	SCS
	AIX 4.2	1 :1	POSIX Draft 7	SCS
	AIX 4.1	1 :1	POSIX Draft 7	SCS
	AIX 3.5	N :1	DCE threads	PCS
	OS/2	1 :1	OS/2 threads	SCS
SGI	IRIX 6.5	M :N	POSIX 1003.1c	SCS, PCS
	IRIX 6.2	M :N	POSIX 1003.1c[3]	SCS, PCS
	IRIX 6.1	1 :1	POSIX 1003.1c	SCS
Linux	Linux 2.x	1 :1	POSIX 1003.1c	SCS
	Linux >1.2	N :1	voir note 4	PCS
Microsoft	Windows NT	1 :1	Win32 threads	SCS
	Windows 98	1 :1	Win32 threads	SCS
	Windows 95	1 :1	Win32 threads	SCS

Comme nous avons déjà mentionné auparavant, les *threads* POSIX sont répandues de telle façon que la plupart des constructeurs offrent une implantation sur leur systèmes UNIX. Néanmoins, l'implantation du standard est parfois incomplète et diffère de système à système. Ceci, provient en partie du caractére optionnel de certains traits de la norme.

Un autre point de divergence provient du fait qu'avant la conclusion du standard POSIX *threads*, des bibliothèques ont été implantées à partir d'ébauches du standard. La plus connue est celle du consortium OSF : **DCE threads**. La bibliothèque DCE est basée sur une version préliminaire du standard, le *draft* originalement référencée comme 1003.4a. Après, ce *draft* a évolué vers les *drafts* 7 et 8 pour finalement aboutir au *draft* 10 qui définit au standard POSIX *threads* (1003.1c) tel qu'il est actuellement.

Le tableau 2.1 fait une comparaison entre les différentes caractéristiques présentées par les implantations du standard POSIX *threads* des principaux constructeurs informatiques.

2.5 Les librairies et la multiprogrammation légère

Dans une application développée en terme de *threads*, celles-ci peuvent accéder concurremment aux ressources et aux fonctions qui composent un processus. De telles applications doivent prendre en considération cet aspect pour garantir l'intégrité des données et des ressources du système. Pour cela le standard POSIX 1003.1c définit une série de termes pour caractériser le comportement d'une fonction ou même d'une librairie vis à vis des *threads* :

- **async-cancel safe :** une fonction peut être appelée par un *thread* quand le mécanisme d'annulation (*cancellation*) est activé ;
- **async-signal safe :** une fonction qui peut être exécutée sans restriction à partir de routines traitantes de signaux ;
- **fonction réentrante :** une fonction est réentrante quand le résultat d'un appel effectué par un ou plusieurs *threads* présente le même comportement que lorsqu'elle est exécutée soit par un *thread* après l'autre dans un ordre indéfini, soit d'une façon entrelacée ;
- **thread safe :** une fonction qui s'exécute correctement et qui maintient un état consistent lorsqu'elle est exécutée par deux ou plusieurs *threads* concurrents. Le terme **MT-safe** est un synonyme ;
- **thread unsafe :** quand l'exécution par plusieurs *threads* donne des résultats incorrects ou imprédictibles. Le terme **MT-unsafe** est un synonyme ;

La définition POSIX de *thread-safe* n'est pas complète car elle ne considère pas l'ordonnancement des exécutions. Une fonction peut avoir un comportement correct lorsqu'elle est appelé par plusieurs *threads* concurrement (*MT-safe*), mais celui-ci peut ne pas correspondre au comportement attendu. Par exemple cette function peut exécuter un appel système bloquant et ainsi bloquer tous les autres *threads* alors qu'un blocage du *thread* exécutant la fonction aurait été suffisant. C'est le cas d'une bibliothèque d'E/S faisant des appels synchrones avec une implantation N :1 ou N :M avec M petit devant N. On peut se trouver avec aucun *thread* actif alors qu'il y en a encore des prêts à s'exécuter. Pour être clair sur cet aspect, nous utilisons le terme **thread aware**. Une fonction **thread aware** est donc une fonction *thread-safe* où l'avancement de *threads* est garanti indépendemment du modèle d'implantation de *threads* utilisées.

2.6 Évaluation de performance

Tout au long de ce chapitre nous avons évoqué plusieurs fois le mot « performance ». Cependant le standard POSIX 1003.1 ne spécifie aucune procédure ni indicateurs pour évaluer les primitives de la multiprogrammation légère. C'est autant

plus grave que la plupart des implantations reposent sur des ordonnanceurs de caractéristiques inconnues. Dans l'annexe G du standard POSIX sont précisés pour les développeurs de librairies de *threads* et d'applications, quelques indicateurs caractéristiques de l'efficacité. Ces indicateurs représentent les efforts de la commission « IEEE P1003.4 working group committee » chargée des aspects relevant du temps réel.

Dans la pratique ces indicateurs de performance sont très difficiles à mesurer car certaines mesures reposent sur l'existence et la maîtrise des priorités, ce qui n'est pas toujours vrai dans les librairies existantes. Un autre point est l'interprétation des résultats de ces tests : sont ils représentatifs de vraies applications ? peut-on les utiliser pour prédire le comportement et/ou la performance d'applications réelles ?

A notre connaissance, aucun consensus sur un jeu de tests pour les libraires de *threads*, ni aucune méthodologie pour les créer ne se sont dégagés au sein de communauté informatique[9]. Certains auteurs proposent leur propre suite de tests [30][112][150]. Des propositions ont été faites sur la liste de discussions sur le WEB (*newsgroup*) *comp.programming.threads*. La plupart de ces tests mesurent le coût de création des *threads*, les durées d'exécution des primitives de synchronisation et le temps de commutation de contexte.

2.6.1 Indicateurs de performance

Nous présenterons dans les 2 sections suivantes les indicateurs le plus courament cités et leur évaluation sur une implantation POSIX *threads* pour le système Solaris.

Coût de création d'un *thread*

Algorithme 2.1 Estimation du coût d'utilisation d'un *thread*

```
1:  f1(n) {
2:    exit;
3:  }
4:
5:  EvalueTemps {
6:    t0 ← GetTime()
7:    create(f1);
8:    join(f1);
9:    t1 ← GetTime();
10: }
```

Mesurer le temps de création d'un *thread* de façon « portable » n'est pas trivial car le comportement des noyaux diffèrent entre eux selon leurs implantations (type

[9] Au moment de l'écriture de ce document (septembre, 1999).

d'ordonnanceur, priorités). Mesurer simplement le temps de l'appel de la primitive de création (*pthread_create()*) n'est pas correct car sur certains noyaux une partie de l'initialisation du *thread* est faite au moment où il est activé pour la première fois. Une méthode « portable » consiste à mesurer le coût d'utilisation global d'un *thread* c'est à dire son cycle de vie [30][112]. Ce coût est obtenu en créant un *thread* qui exécute une procédure vide, le créateur attend sa terminaison via une primitive *join*. Le temps mesuré correspond à une prise de temps avant l'appel *pthread_create()* et après *pthread_yield()*. Cette procédure est donnée par l'algorithme 2.1.

Commutation de contexte

Ce test pose aussi un problème d'évaluation à partir d'un programme utilisateur car on ne peut pas forcer directement une commutation de contexte. La procédure de « passage de main » (*yield()*), d'une part ne figure pas au standard POSIX, d'autre part a souvent une implantation qui ne provoque pas nécessairement un commutation de contexte[10].

Pour estimer ce coût, nous avons utilisé 2 *threads* se bloquant alternativement par de séquences P&V croisés (interface UI, Solaris 2.6). Le temps de commutation est obtenu en mesurant le temps écoulé entre le début et la fin de la séquence. Nous avons oté de ce temps le temps nécessaire pour exécuter une opération P&V non bloquante. En fait, ce temps représente $2 \times n$ commutations de contexte. Il est clair que la mesure comprend non seulement le temps de commutation de contexte mais aussi celui de la gestion de file d'attente du sémaphore. Cela n'est pas génant dans la mesure où toutes les opérations de synchronisation incluent un tel coût. L'algorithme 2.2 présente la procédure réalisée.

L'emploi de sémaphores de l'interface UI de solaris 2.6 est dû au fait qu'ils nous permettent de garder la même structure pour évaluer le temps de commutation entre processus lourds. Cette méthode, réposant sur une caractéristique particulière, n'est pas portable. L'algorithme 2.3 implante la même philosophie de deux *threads* que se bloquent mutuellement n'utilisant que des appels aux primitives POSIX *threads* donc portable. Ici le blocage est effectué à l'aide des variables de condition. On crée deux *threads* qui en fonction de leur identification exécutent une partie distincte de la structure de contrôle *if* (ligne 6). Le *thread 0* attend sur une condition, et le *thread 1* la signale. A chaque tour, les rôles sont inversés. A la fin d'un certain nombre d'itérations (N) la boucle est rompue, $2 \times N$ commutations de contexte ont été réalisées. Le temps obtenus par cette méthode sont équivalents à ceux du premier algorithme.

[10]Nous avons vérifié cela avec l'aide de l'outil de visualisation de l'environnement ATHAPASCAN (Pajé) sur le système Solaris 2.6. C'est aussi le cas sur linux.

Algorithme 2.2 Estimation du temps de commutation de contexte (P&V)

```
1:  Thread_1(n) {
2:  t0 ← GetTime()
3:  V(sem0)
4:  for i = 0 to n do
5:     P(sem1)
6:     V(sem0)
7:  end for
8:  t1 ← GetTime();
9:  }
10:
11: Thread_2(n) {
12: for i = 0 to n do
13:    P(sem0)
14:    V(sem1)
15: end for
16: P(sem0)
17: }
```

Algorithme 2.3 Estimation du temps de commutation de contexte (POSIX threads)

```
1:  Thread(tid) {
2:  count ← 0
3:  t0 ← GetTime()
4:  lock(m)
5:  while count < N do
6:     if tid ≠ 0 then
7:        tid ← 0
8:        cond_signal(c)
9:     else
10:       tid ← 1
11:       count ← count + 1
12:       cond_wait(c,m)
13:    end if
14: end while
15: cond_signal(c)
16: unlock(m)
17: t1 ← GetTime()
18: }
```

Coût de synchronisation

Ici, on mesure le surcoût de la synchronisation c'est à dire quand un blocage n'est pas nécessaire. Il est obtenu par l'exécution d'une boucle contenant la primitive à évaluer. Deux cas sont à évaluer. Dans le premier, l'objet de de synchronisation, verrou ou sémphore, est toujours non bloquant. Ainsi nous obtenons le coût des primitives *lock/unlock*, *trylock/unlock* et P&V. La deuxième situation est celle concernant la primitive *trylock* c'est à dire le test et prise du verrou s'il n'est pas pris. L'algorithme 2.4 illustre ce principe.

Algorithme 2.4 Estimation du coût minimum de synchronisation

```
1: EvalueTemps {
2: t0 ← GetTime()
3: for i = 0 to n do
4:    lock(m)
5:    unlock(m)
6: end for
7: t1 ← GetTime();
8: }
```

2.6.2 Expérimentations

Le tableau 2.2 donne les résultats obtenus lors de l'exécution des tests ci dessus en utilisant le noyau POSIX *threads* fourni par le système Solaris 2.6. Pour donner une référence, nous avons mesuré le coût de création de processus « lourd » pour clonage (*fork*).

2.7 Bilan

Le concept de *thread*, la technique de multiprogrammation légère et leurs implantations (1 :1, N :1, M :N) fournissent des outils appropriés pour le développement d'applications concurrentes. Notre intérêt se porte sur les implantations 1 :1 et M :N car elles permettent l'exploitation du parallèlisme des architectures multiprocesseurs. De plus, la multiprogrammation permet le recouvrement des délais de communication par du calcul. C'est donc un outil privilégiée pour la programmation parallèle sur des grappes de multiprocesseurs interconnectéss par un réseau.

Différentes implantations de bibliothèques de *threads* existent mais leurs fonctionnalités de base sont très voisines. On notera cependant qu'il n'y a pas d'accord sur les politiques d'ordonnancement qui restent souvent imprécises et incomplétes. Peut être du fait que le standard POSIX est assez « tolérant » à ce sujet.

Tableau 2.2 *Temps des principales primitives POSIX* threads
(Caiapo : solaris 2.6, pentium @133Mhz ; Guarani : solaris 2.6, bi-pentium II @333Mhz ; Comanche : solaris 2.6, quadri-pentium pro @200Mhz ; Huron : solaris 2.6, bi-sparc @450Mhz)

	Caiapo	Guarani	Huron	Comanche
Création processus	23.5 ms	8.6 ms	9.0 ms	12.5 ms
Création thread(noyau)	457μs	156μs	145μs	207μs
Création thread(utilisateur)	110μs	32.6μs	48μs	57.5μs
chang. contexte processus	67μs	15μs	15μs	21μs
chang. contexte(noyau)	66μs	12.5μs	14.64μs	18.8μs
chang. contexte(utilisateur)	12.7μs	3.65μs	6.05μs	6.16μs
lock/unlock(POSIX threads)	0.700μs	0.325μs	0.281μs	0.534μs
trylock/unlock(POSIX threads)	0.714μs	0.325μs	0.281μs	0.534μs
trylock(verrou non disponible)	0.350μs	0.160μs	0.180μs	0.270μs

Aucun consensus n'existe sur des tests de réference de mesures de performance. C'est pourquoi le portage d'une application parallèle exige toujours une phase d'évaluation et d'adaptation à la nouvelle architecture. C'est particulièrement le cas pour ATHAPASCAN-0 qui exige un couplage serré entre un noyau de *threads* et la bibliothèque de communication MPI qui ignore la notion de *thread*. Le chapitre suivant est dévolu à la présentation du standard de communication MPI fixant les fonctions d'une telle bibliothèque.

3

Communication par message : le standard MPI

« Gardez-vous de rien dédaigner, surtout quand vous avez à peu près votre compte. » Jean de La Fontaine, Le héron

Dans ce chapitre nous allons présenter le standard MPI (Message Passing Interface) qui est le plus utilisé pour le développement d'applications parallèles dans un environnement distribué. Initialement, nous introduirons les concepts de base et la terminologie associée au paradigme de programmation par échange de messages. Ensuite nous parlerons du standard MPI, ses caractéristiques et son évolution. Pour conclure nous présenterons un modèle et un ensemble de métriques couramment employés pour évaluer le comportement d'une librairie de communication comme celle proposée par le standard MPI.

3.1 Principe de programmation par échange de message

Le parallélisme consiste essentiellement à diviser un problème donné en plusieurs sous problèmes et à les résoudre simultanément. L'expression de ce parallélisme peut être plus au moins explicite. Dans le cas d'un parallélisme implicite, un compilateur assure l'extraction du parallélisme du programme et génère le code effectuant le placement des calculs et données sur les processeurs, les échanges de données et synchronisation nécessaire entre calculs. A contrario dans un modèle où tout doit être explicite, le programmeur doit explicitement décider du placement des proces-

sus, des partages de données et des synchronisations. Comme nous avons vu dan le chapitre précédent ces coopérations peuvent se faire via une mémoire commune.

En l'absence de mémoire commune il est nécessaire d'apparier données et processus, la coopération et la synchronisation entre eux se fait par l'« échange de message » (*message passing*). Un programme parallèle est donc vu comme un **ensemble de processus coopérants**. Ces processus peuvent être exécutés simultanément sur différentes machines, ou concurremment sur une seule machine, exploitant les mécanismes classiques de systèmes d'exploitation c'est à dire, création et destruction des processus lourds, ordonnancement, temps partagé, etc. Étant donné qu'il n'y a aucune restriction à propos du nombre de processus par processeur on parle de **machine parallèle virtuelle** [72][167]. Les processus ne partagent pas de mémoire. La coopération ne se fait que par l'émission et la réception de message. Cela correspond à une copie explicite d'une donnée de la mémoire de l'émetteur vers celles des récepteurs. A une communication par message est toujours attachée une synchronisation provenant du fait qu'une réception ne peut être faite qu'à partir du moment où la zone de réception a été fournie par le récepteur. Cette contrainte se propage à l'émetteur (contrôle de flux). Les différents modèles communicants différent selon les caractéristiques de la création des processus et leur implantation ainsi que par la sémantique des opérateurs de communication.

3.1.1 Les processus

Fondamentalement, il existe deux méthodes pour la création de processus c'est à dire la configuration de la machine parallèle virtuelle : **statique** ou **dynamique**. Dans la méthode *statique*, tous les processus nécessaires à l'évaluation du problème sont créés lors du lancement de l'application. La méthode *dynamique* considère que les processus peuvent créer d'autres processus en cours d'évaluation. Ici aussi la définition précise de ce qu'est un processus fixe le coût de création. Les premières implantations du modèle communicant (MPI, PVM) avaient une implantation lourde, c'est à dire qu'un processus était l'exécution d'un fichier exécutable. Cependant du fait de l'existence de mémoires différentes, il était possible aux processus d'exécuter ou non des copies du même fichier.

Dans le modèle SPMD, *Single Program Multiple Data*, les processus exécutent des fichiers exécutables correspondant au même programme. Au moment du lancement de l'application, chaque machine charge une ou plusieurs copies du programme. Si les machines sont hétérogènes, le programme doit être compilé pour chaque machine cible. Le principal désavantage de ce modèle est le gaspillage de mémoire puisque chaque processus contient une copie complète du programme pour en exécuter seulement une partie. Le grand avantage est sa simplicité de gestion.

Dans le modèle MPMD, *Multiple Program Multiple Data*, les processus exécutent des programmes différents les uns des autres. Chaque processus exécute un

code spécifique pour accomplir la tâche qui lui a été affectée. Le démarrage d'un modèle MPMD peut être fait à partir d'un programme initial qui exécute et crée les autres processus.

La création d'un processus nécessite en plus de la donnée du code à exécuter, le transfert des arguments mais surtout la localisation du processeur (station, etc) devant l'exécuter (placement). La création d'un processus est une action à distance nécessitant de communiquer.

3.1.2 La communication

On reconnaît deux classes d'opérateurs de communication selon les interlocuteurs impliqués. On parle de communication point à point (bipoint) s'il y a un émetteur et un recépteur sinon on parle de communication collective (multipoint).

Les primitives de communication point à point

Deux opérateurs de base *send()* et *recv()* permettent à une paire de processus - l'émetteur et le destinataire - d'échanger des messages entre eux. Le nombre et la complexité des paramètres de ces primitives varient selon la bibliothèque d'échange de messages utilisée. Cependant ils doivent permettre de résoudre les problèmes suivants :

- identification des interlocuteurs ;
- description des zones mémoire à émettre et des zones mémoire pour la réception ;
- garantie d'un ordre de délivrance des messages ;

Dans le plus simple des cas, on considère qu'il existe un canal unidirectionnel entre toute paire de processus. Sur un canal, les messages sont acheminés dans l'ordre d'émission (canal FIFO). Du point de vue de l'émetteur ou du récepteur, ce canal est simplement identifié par le processus interlocuteur. C'est la solution choisi par la plupart des bibliothèques de communication. Cette solution s'oppose à celle identifiant explicitement le canal (e.g. socket dans les protocoles IP). Si deux processus utilisent des flots de messages distincts, il faut pouvoir gérer plusieurs canaux logiques entre ces processus. Ceci est fait simplement en étiquettent les communications (*tag*) par le numéro de canal logique à utiliser entre 2 processus.

```
send( message, type, quantité, étiquette, dest)
recv( message, type, quantité, étiquette, source)
```

Toutefois l'identification explicite d'un émetteur par la primitive *recv* peut poser des problèmes. Prenons, par exemple, la situation où un processus est serveur pour d'autres processus. Ce serveur doit attendre les requêtes des processus clients. Étant

donné l'indépendance de ceux-ci, aucun ordre pour la réception ne peut être prévu a priori. Par conséquence le processus destinataire (le serveur) doit pouvoir recevoir un message d'un émetteur quelconque (le client) pour éviter des situations de blocage. La solution la plus simple consiste à pouvoir recevoir un message de n'importe quel processus sur n'importe quel canal[1]. Une forme plus sophistiquée consiste à définir l'ensemble des canaux écoutés (e.g. l'appel *select* d'UNIX).

La définition de ces procédures reste incomplète tant que l'on n'a pas précisé les synchronisations attachés à ces opérateurs. Celles-ci dépendent des protocoles de communication utilisés et des interactions entre les démons les assurant et les processus les demandant[152]. Ainsi les requêtes d'émission ou réception peuvent être **bloquantes** (synchrones) ou **non-bloquantes** (asynchrones). Dans une requête bloquante d'émission, le processus est bloqué jusqu'à ce que le message soit complètement émis vers le processus destinataire, ou bien mémorisé par la couche de communication. Ceci veut dire que la zone d'émission peut être réutilisée à nouveau. Pour une opération de type *receive* (bloquant) le processus qui l'exécute est bloqué jusqu'à l'arrivée du message.

Un opérateur non bloquant enregistre simplement l'opération à faire ultérieurement par la couche de communication. Le principal avantage est une gestion précise du recouvrement[2] du calcul par les communications par le processus puisque celui ci peut continuer son exécution en même temps que l'acheminement de la communication. Il est nécessaire de rajouter un moyen de tester et/ou attendre la fin d'une requête soit pour réutiliser une zone d'émission soit pour savoir si une donnée a été reçue. Ceci peut se faire par des opérateurs de test d'attente explicite ou par l'utilisation de fonctions réflexes (*call back*) appelées par la couche de communication à la fin d'une requête.

Restent les synchronisations attachées à la façon dont la couche de communication effectue le contrôle de flux. L'efficacité des communicateurs est conditionnée par le choix de la méthode. Le problème du contrôle de flux se pose dès qu'un message arrive chez un destinataire avant qu'il ait exécuté la primitive *recv* correspondante. Normalement, un des paramètres de la primitive *recv* est justement l'adresse d'une zone mémoire pour la réception du message. Que faire si cette adresse n'est pas encore définie ?

Une des stratégies est d'ignorer l'arrivée du message et de définir un protocole de retransmission. Ceci suppose que le message ait été conservé chez l'émetteur. Une autre est de stocker ce message dans un tampon intermédiare jusqu'à ce que la primitive *recv* soit exécutée par le destinataire. La première solution impose un surcoût de gestion d'un protocole de récupération et retransmission du message. La deuxième, nécessite une structure de données pour stocker temporairement les messages (les tampons[3]) jusqu'à l'exécution de la primitive *recv* qui permettra la

[1]Des valeurs particulières de processus et d'étiquette *any_source* et *any_source* précisent cette situation.

[2]overlapping.

[3]En anglais, buffers.

copie du tampon vers la zone définitive. Le coût, en plus de la copie, est celui de la gestion de tampons : allocation, libération, insertion, etc. De plus, cette deuxième stratégie n'élimine pas complètement le problème de la réception car les tampons ne sont pas en nombre illimité. Cette technique retarde simplement l'occurrence du problème.

Certains systèmes mettent en place une troisième option : le processus n'émet pas de message tant que le destinataire n'a pas fait le *recv* correspondant. Dans ce cas, un protocole est nécessaire pour signaler à l'émetteur que le récepteur est prêt. Cette solution élimine le besoin de tampons mais ramène le coût d'une communication à un aller/retour dans le réseau.

Une solution intermédiare consiste à utiliser suffisamment de tampons pour recevoir des petits messages ou des descripteurs de gros messages (enveloppe). L'envoi d'un gros message est fait en deux parties : une « intention » d'envoi, l'enveloppe, et l'envoi du message proprement dit. L'émetteur envoie donc l'**enveloppe**. A la réception de l'acquitement de cette intention, émise lors de l'exécution de l'opération *recv* chez le destinataire, l'émetteur envoi le message.

Les primitives de communication collectives

Les bibliothèques de communication offrent aussi un ensemble de primitives pour la communication entre processus d'un groupe. Essentiellement, les primitives de communication collectives disponibles dans la plupart des bibliothèques sont de deux types : **diffusion** et **réduction**.

Dans la *diffusion*, nous trouvons différentes variations. La première est la **diffusion totale** (broadcast). Ici un processus, dit racine (*root*), appartenant à un groupe, envoie un message à tous les autres processus participants au groupe. Le processus racine peut être choisi parmi ceux qui composent le groupe. La deuxième version est la **distribution** (scatter) qui consiste à répartir les données d'un tableau présent sur un processus (la racine) sur les autres processus du groupe. Le contenu du n^e position du tableau est envoyé au n^e processus du groupe. L'opération de **regroupement** (gather) est l'inverse de la *distribution* et permet à un processus de récupérer un ensemble de données à partir d'autres processus. Essentiellement, une donnée du n^e processus est reçu par le processus racine et mise sur la n^e position d'un tableau destiné à la réception de données. Finalement, l'**échange total** (*scatter/gatter*) où les processus d'un groupe envoient et reçoivent les données de tous les autres processus participants du groupe.

Les opération de type **réduction** sont en fait une variation de l'opération de *regroupement* où les valeurs reçues par le processus racine sont combinées en une seule valeur par le biais d'une opération associative et commutative.

Le problème posé par ces primitives est celui d'une implantation efficace mais aussi celui de la gestion de groupe. En effet ces opérateurs terminent quand tous les processus ont émis et reçus les données nécessaires. Ceci suppose que la compo-

sition d'un groupe soit connu de tous les processus. Dans le cas où les groupes se créent et évoluent dynamiquement, la gestion d'un état cohérent d'un groupe est un problème difficile en général. Dans le cas de la programmation parallèle, on accepte des restrictions pour la création d'un groupe : il faut le créer à partir de groupes existantes. Tous les membres des groupes concernés doivent « agréer » cette création. Ce consensus est établi par une opération de type barrière qui assure que tous les processus soient d'accord. Il existe toujours un groupe initial représentant tous les processus. Naturellement la création dynamique de processus nécessite l'agrément de tous les autres processus. C'est pourquoi les bibliothèques offrent des opérations collectives ne permettent généralement pas la création dynamique des processus.

3.1.3 Les bibliothèques de communication

La programmation par échange de messages est largement utilisée sur des machines à mémoire distribuée. Plusieurs bibliothèques ont été mises à disposition soit par les constructeurs de machines soit par des laboratoires de recherche, comme par exemple, MPL [13], Express [108], Parmacs [31], Intel's NX [128], NCube's vertex [121], P4 [113], Zipcode [145], etc. Initialement la plus répandue a été PVM [85], acronyme pour *Parallel Virtual Machine*, développée par le Oak Ridge National Laboratories (USA). Le succès de PVM était dû, d'une part, au fait qu'il proposait un environnement de programmation par échange de messages pour des milieux hétérogènes et homogènes ainsi qu'une collection de primitives pour des applications écrites en C et en Fortran. D'autre part, accessible gratuitement à partir du WEB, PVM était disponible pour une grande gamme de machines et de systèmes d'exploitation.

Pour éviter la prolifération de différentes versions de bibliothèques de communication, et corriger des insuffisances de l'interface de PVM, un groupe d'universités et de constructeurs informatiques ont défini un « standard » de communication par messages : MPI (*Message Passing Interface*) [147].

3.1.4 Implantation des bibliothèques de communication

Les environnements existants de programmation parallèle par passage de message tels que PVM et MPI ont une implantation lourde. Les processus sont implantés par les processus lourds des systèmes hôtes comme nous l'avons vu précédemment (section 3.1.1). L'implantation de la communication peut être assurée intégralement au sein d'une bibliothèque liée à tous les processus de l'application ou bien celle ci délègue une partie du travail à un processus lourd spécialisé : le démon de communication.

La communication entre le processus utilisateur et le démon de communication est fait en utilisant les IPC (Inter Process Communication) existants sur les systèmes d'exploitation hôtes. La figure 3.1.a montre cette organisation. Les démons

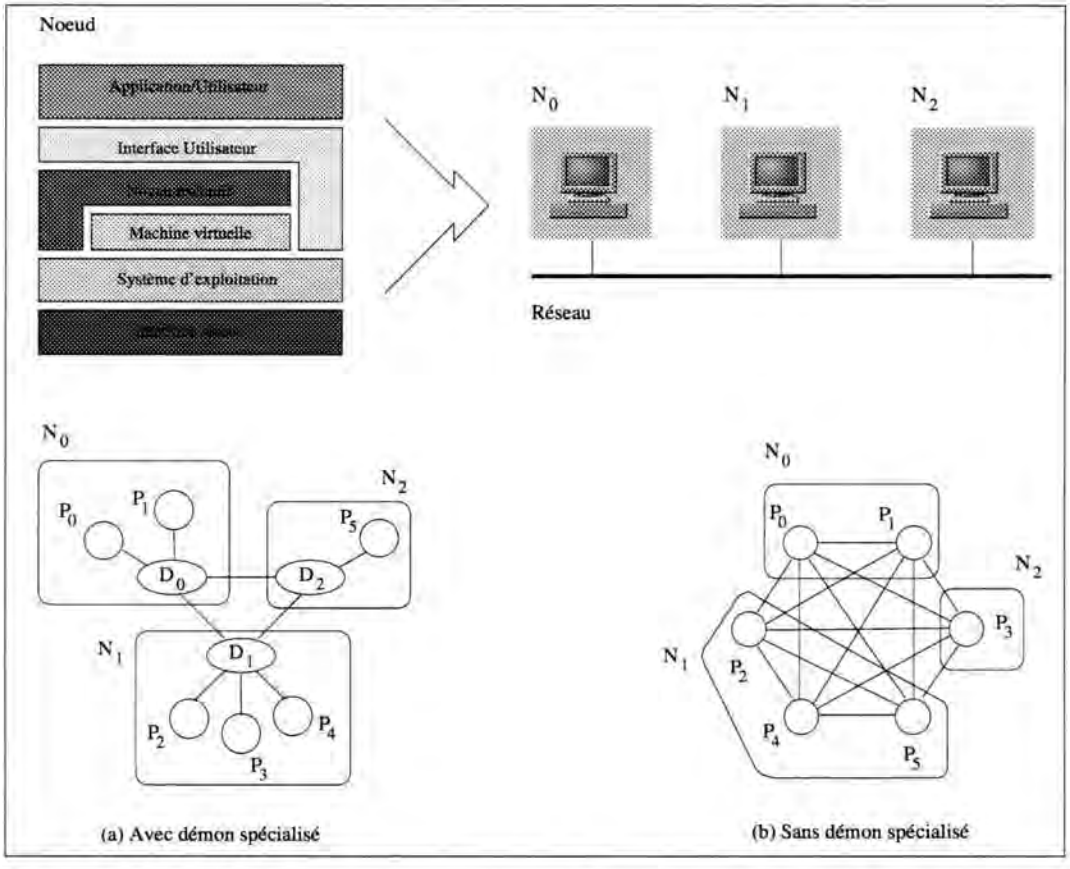

Figure 3.1 *Implantations des environnements de programmation par échange de messages*

communiquent entre eux pour créer les processus à distance, émettre et recevoir les messages de/vers d'autres nœuds. L'intérêt d'utiliser un processus démon est :

- un contrôle plus simple de l'évolution dynamique du nombre de processus composant la machine virtuelle (par inclusion ou suppression) ;
- avoir un processus dédié à l'écoute du réseau et des processus de calcul simplifie la gestion de l'avancement des communications. Sans démon, cet avancement doit être assuré par la bibliothèque ce qui en complique fortement la réalisation ;
- réduire le nombre de connexions réseau à gérer par nœud à $n-1$ où n est le nombre de nœuds plutôt que $p-1$ où p est le nombre total de processus.

Le plus grand inconvénient de cette approche est le surcoût de recopie des messages entre processus utilisateur $\leftrightarrow$ processus démon. Ce coût varie selon le mécanisme IPC employé : mémoire partagée, canaux (*pipe*), file de messages, etc. Ces surcoûts peuvent être éliminés par une liaison directe entre processus utilisateurs (fig. 3.1.b).

On trouve ces deux types d'implantation pour les différents interfaces de communication. Par exemple, PVM initialement utilisait un démon pour faire des communications. Dans les versions plus récentes les processus communiquent directement. MPI s'est dès le départ partagé entre des implantations avec (e.g. LAM[4]) et sans démon (e.g. MPICH).

3.2 Le standard MPI : Message Passing Interface

Le MPI [147], acronyme de *Message Passing Interface*, est un standard issu des efforts conjoints de constructeurs de machines parallèles, d'utilisateurs, de laboratoires de recherche et d'universités, afin de définir la syntaxe et la sémantique d'un ensemble de routines destinées au développement de programmes parallèles basés sur le paradigme d'échange de messages. Ce consortium a donc décidé d'intégrer les fonctionnalités les plus intéressantes déjà existantes sur plusieurs systèmes et d'en combler les éventuelles lacunes, comme par exemple, la description de données complexes en mémoire. Le résultat a été un ensemble riche et flexible de fonctions pour l'échange de messages. Le spectre de ces fonctions (plus de 120) s'étend des primitives point à point aux primitives collectives, jusqu'à la définition de types composés, de groupe de processus, de domaines de communications, de topologies de connexion entre processus, une interface pour la prise de trace et l'instrumentation[5], la gestion de l'environnement d'exécution. Ces primitives sont offertes en langage C et Fortran77.

Néanmoins, des points importants pour le développent de programmes parallèles n'y figurent pas : (a) les opérations d'accès à une mémoire distante ; (b) les messages actifs ; (c) la gestion des machines et des tâches ; (d) les opérations d'entrée sortie ; et (e) le support explicite de la multiprogrammation légère. La définition du standard MPI-2.0 [118] a complété certains points en particulier ceux relatifs au temps réel (MPI-RT). Nous allons donc présenter brièvement le standard MPI en discutant ses concepts de base, et nous conclurons sur certains ajouts de MPI-2.0.

3.2.1 Concepts de base

Le standard MPI définit donc une sémantique et un ensemble de concepts de base pour la programmation par échange de messages. MPI ne spécifie aucun opérateur pour la création dynamique de processus. Les processus sont statiquement créés à l'initialisation. Ils peuvent exécuter un même programme (mode SPMD) ou différents programmes (mode MPMD). Dans cette section nous présentons les concepts de base et la terminologie définie par le standard MPI.

[4]En effet, LAM offre les deux types de fonctionnement par le biais d'options de lancement.
[5]Profiling.

Processus/Groupe : Une application MPI est donc un ensemble statique[6] de processus appelé *world*. A chaque processus est associé un identificateur (*rank*), numéroté de 0 à $n-1$ où n est la cardinalité de l'ensemble *world*. Ce groupe initial peut être subdivisé en plusieurs sous groupes pendant l'exécution du programme. Un processus peut appartenir donc simultanément à plusieurs sous groupes. Des primitives pour déterminer le *rank* et le nombre de processus (*size*), appartenant à un groupe (sous groupe) sont disponibles.

Les groupes sont toujours des sous ensembles de l'ensemble initial *world*. Les opérateurs de construction de nouveaux groupes par partition ou fusion de groupes préexistants sont des opération synchrones de type barrière qui nécessitent l'accord de tous les membres des groupes qui se partitionnent ou se fusionnent. C'est ainsi que MPI garantit une gestion cohérente de l'état des groupes et de leur désignation.

Un groupe est identifié par un communicateur[7]. Un processus sera donc toujours désigné par un rang (*rank*) relativement au groupe représenté par le communicateurs.

Communicator : Les processus peuvent appartenir à plusieurs groupes. Un groupe représente en fait un plan de communication distinct de celui d'un autre groupe même si les interlocuteurs sont les mêmes. Ceci justifie le choix du nom « communicator ». Ce concept a été introduit pour éviter de mélanger les communications d'un calcul parallèle avec celles des bibliothèques parallèles appelés par les processus exécutant le calcul. On peut considérer qu'il y a un communicateur par bibliothèque. Dans un plan, les communications MPI ne présentent pas d'originalité particulière. Les processus communiquent via des liaisons point à point FIFO. Un processus les désignent par une étiquette (*tag*) et l'identificateur de l'interlocuteur c'est à dire un rang dans un communicateur. Les opérations collectives concernent tous les processus associés à un communicateur.

Le *communicator* est un paramètre obligatoire pour toutes les primitives de communication.

Sémantique des primitives de communication point à point : Le standard MPI définit des primitives de communications non bloquantes (*non blocking*) et bloquantes (*blocking*). Une primitive non bloquante est celle qui initie simplement une opération de communication. Ces primitives ont un argument additionnel appelé **requête** (*request*), lequel est utilisé par d'autres primitives pour vérifier la terminaison d'une opération non bloquante précédemment démarrée. L'utilisation de telles primitives permet, si c'est possible, un recouvrement des calculs et des communication. Le terme *bloquant*, pour le standard MPI signifie qu'au retour d'une primitive de ce type, le tampon associé aux données peut être réutilisé (retour d'un *send*) ou contient des données consistantes (retour d'un *recv*).

[6]Comme le définit le standard MPI-1. Toutefois l'implantion LAM de MPI supporte la création dynamique de processus par le biais de l'appel MPI_Spawn. La création dynamique de processus est partie intégrante de la définition de MPI-2.

[7]Un communicateur est plus général qu'un groupe car on peut lui attacher des informations diverses spécifiques de MPI ou de son utilisateur.

MPI définit quatre modes d'acheminent des données entre émetteur et récepteur. Les modes sont :

- **Buffered :** ce mode présent une sémantique de terminaison dite locale c'est à dire la terminaison de l'opérateur *send* est indépendante du moment où le *recv* correspondant est fait. Le message à émettre est stocké localement chez l'émetteur. L'émission termine alors.
- **Standard :** dans ce mode l'implantation est libre de choisir si l'opérateur *send* réalise ou non le stockage des messages à émettre. Si le stockage est fait, un appel *send* peut terminer avant que le *recv* associé soit posté. S'il n'y a plus de ressources pour le stockage (tampons), ou que l'implantation ne le réalise pas, le *send* n'est terminé qu'après que les données soient transférées chez le récepteur. La sémantique de terminaison est dite non locale. Le standard n'impose pas de contrôle de flux. Ainsi certaines implantations (MPICH, MPI-IBM) n'implantent pas de contrôle de flux et arrêtent l'exécution en cas de saturation des tampons du destinataire.
- **Ready :** pour éviter ce problème, le mode *ready* n'émet pas le message tant que le *recv* n'a pas été fait. C'est une synchronisation de type rendez-vous. Le *send* termine quand le rendez-vous est terminé.
- **Synchronous :** ce mode a une sémantique de completion non locale. L'émission a lieu aussitôt que possible et considère que le *recv* correspondant a été posté préalablement. Dans le cas contraire, le comportement n'est pas défini (perte, erreur, etc).

Primitives de communication collectives : MPI permet, pour le processus appartenant à un même groupe, de réaliser des opérations de diffusion, regroupement, distribution et réduction. Ces primitives doivent être exécutées par tous les processus du groupe affecté (contrairement à des primitives point à point, la valeur de l'étiquette (*tag*) n'est pas prise en compte). Une primitive de synchronisation de type barrière (*barrier*) est aussi offerte.

Définition des types de données : MPI a été conçu pour des machines hétérogènes pour lesquelles la communication peut exiger des transcodages de données. Par ailleurs, la communication entre processus peut impliquer le transfert de données complexes non nécessairement contiguës en mémoire. Pour assurer les transcodages de façon transparente au programmeur et automatiser l'emballage et le déballage de données complexes, MPI définit les types élémentaires de données communicantes (vecteur d'entiers, flottants, etc) et des constructeurs de type permettant de décrire l'implantation en mémoire de données non contiguës. Le programmeur peut utiliser ces fonctionnalités de deux façons. Dans un opérateur de communication l'indication du type permet à MPI d'effectuer l'emballage ou le déballage de la variable. MPI gère donc tous les tampons et copies nécessaires à l'émission (ou la réception), sinon l'utilisateur peut décider d'emballer/déballer lui même les données dans une zone mémoire qui sera communiquée en bloc. Pour cela, il dispose d'opérateurs spécifiques d'emballage/déballage (*pack*, *unpack*).

Topologie virtuelle : La topologie est un attribut optionnel aux communicateurs utilisée pour fournir un mécanisme de désignation des processus appartenant à un groupe. Comme vu précédemment, chaque processus dans un groupe est identifié par un rang entre 0 et $n-1$, n étant le nombre de processus du groupe. Or sur certaines applications parallèles, une numérotation non linéaire est mieux adaptée pour décrire le schéma de communication logique entre ces processus, comme par exemple, sur de grilles (2 dimensions ou 3 dimensions). Le standard MPI définit des fonctions pour assurer les mouvements efficaces de données prenant en compte une organisation spatiale des processus en forme de grille (*cartesian*) et graphe. Ce mécanisme de désignation peut encore, pour une implantation particulière de MPI, aider au placement efficace de processus en respectant la géométrie physique du matériel.

3.2.2 MPI et les processus légers

Le standard MPI-1 **ne spécifie pas** de modèle d'exécution pour un processus MPI : un processus MPI peut être séquentiel pur, ou utiliser les concepts de multiprogrammation légère (*multithreading*). Le consortium n'a pu établir si la prise en compte des *threads* posait des problèmes spécifiques aux appels des opérateurs de communication. Il a donc été simplement recommandée que les implantations soient *thread-safe*. Ceci aurait permis aux *threads* d'un même processus d'appeler MPI de façon concurrente. Lors d'un appel MPI bloquant, seul le *thread* qui l'exécute devrait être bloqué. Pourtant la plus grande partie des implantations MPI ne sont pas *thread aware* ni *thread safe*. En général l'origine de ce problème est l'utilisation d'anciennes librairies *thread-unsafe* (comme par exemple, P4 pour le cas de MPICH) pour l'implantation de MPI.

A notre connaissance aujourd'hui[8] deux constructeurs offrent une version MPI *thread aware* : IBM pour le système AIX 4.2 (MPI-3.0) et Helwett Packard pour le système HP-UX 11.0.

3.2.3 Les extension MPI-2

Le Forum MPI, lors de la définition du standard MPI-1.0, s'est fixé des écheances et des objectifs pour éviter un standard trop large et pour ne pas retarder les éventuelles implantations. Certaines fonctionnalités n'ont pas été intégrées car aucun consensus ne s'est dégagé ou parce qu'elles nécessitaient des études complémentaires. Ces fonctionnalités devaient être introduites comme des extensions ultérieures.

Le standard MPI-1.0 est finalement sorti au mois de mai 1994. À partir de mars 1995 ce standard a été révisé pour clarifier la sémantique de certaines fonctions, ce qui a donné naissance à la version MPI-1.1 en juin 1995. En parallèle (avril

[8]Septembre, 1999.

1995), le Forum MPI a initié les travaux pour la définition des extensions MPI-2.0. Le nouveau standard a été publié au mois d'avril 1997 [118]. Les extensions portent sur la gestion de dynamique de processus, l'utilisation de communications asynchrone (*one sided*), et les E/S.

Gestion dynamique de processus : MPI-1.0 suit un modèle de programme parallèle basé sur un ensemble statique de processus définis au lancement de l'application. La gestion dynamique de processus offert par la bibliothèque PVM s'étant avérée un outil puissant, une gestion similaire a été intégrée dans MPI-2.0.

Entrée et sorties parallèles : Un handicap important de la programmation parallèle est l'absence de mécanismes portables et performants pour la réalisation d'opérations d'entrée/sortie. Les systèmes parallèles de fichiers existants sont basés sur des variations spécifiques et différentes ce qui complique la portabilité d'un programme parallèle accédant aux fichiers de l'interface POSIX. De plus, cette interface n'est pas adéquate pour exprimer une série de fonctions communes aux programmes parallèles, comme l'accès non contigu aux fichiers ou des opérations collectives sur les fichiers [157].

Le standard MPI-2.0 définit alors une nouvelle interface pour les opérations d'entrée/sorties parallèles - connues comme MPI-IO - composées par une série de fonctions destinées à la réalisation d'entrée/sortie asynchrone, des accès à des fichiers par tranches (*strided*), et le contrôle de la répartition de fichiers sur disques durs.

Opérations *one sided* : Les opérations classiques de communication introduisent un synchronisme entre l'émetteur et le destinataire. MPI-2.0 découple les étapes de communication et de synchronisation en intégrant le concept d'opérations d'accès aux mémoires distantes, RMA (*Remote Memory Access*).

Il y a trois primitives RMA : lecture à distance (*get*), écriture à distance (*put*), et une variation d'écriture à distance où la valeur écrite est en faite combinée avec la valeur déjà existante (*remote accumulate*). La cohérence des accès mémoire relève de la responsabilité de l'utilisateur. En d'autres termes la sémantique de cohérence mémoire est de type *weakly coherence* ce qui est une façon détournée de dire qu'il n'y a aucune.

Aide à la programmation et debogage : L'idée est de permettre à une application MPI écrite dans un langage puisse utiliser les bibliothèques, ou un ensemble de fonctions écrites dans un autre langage. Une série de conventions ont été établies à ce niveau pour l'édition de liens de différents langages, les appels de fonctions unifiés, le passage de paramètres, la correspondance entre les types de base, etc. MPI-2.0 étend aussi les syntaxes de appels des fonctions pour supporter des appels C++ et FORTRAN 90.

Un autre point est l'extension des interfaces externes qui consiste à définir une série d'objets destinés au développement de débogueurs et de traceurs pour applications MPI ainsi que à aider la création de nouvelles primitives basées sur celles déjà existantes sur MPI.

Extensions des opérations collectives : MPI-2.0 lève la restriction que seuls les processus appartenant à un même groupe peuvent utiliser les primitives de communication collectives. L'autre extension est l'apparition de versions non bloquantes pour ces opérations.

Multiprogrammation légère : Le standard MPI-2.0 établit les conditions minimales requises pour qu'une implantation de MPI soit considérée comme *thread compliant* et définit le comportement de *threads* par rapport aux appels MPI. Fondamentalement il existe deux propriétés spécifiques : tous les appels MPI doivent être *thread safe* et un *thread* bloqué sur un appel MPI ne doit pas interdire l'avancement d'autres *threads* prêts à l'exécution appartenant au même processus. Ces conditions rejoignent la définition d'une librairie *thread aware* présentée au chapitre 2 (section 2.5).

Le comportement des *threads*, vis à vis des appels MPI, est définit par une nouvelle fonction d'initialisation (*MPI_Init_thread*) qui spécifie le niveau souhaité de support des *threads*. Le plus simple est le niveau *single*, où un seul *thread* de contrôle est admis. Cela correspond à une utilisation classique de MPI. Si lorsque le processus utilise plusieurs *threads*, tous les appels MPI sont dues à un seul *thread*, nous avons le comportement dit *funneled*. Au troisième niveau *serialised* tous les *threads* sont autorisés à appeler MPI à condition que ces appels soient faits et exécutées[9] l'un après l'autre. Enfin, le niveau *multiple* autorise tous les *threads* à appeler MPI sans aucune restriction. Le standard MPI 2.0 précise encore que les messages ne sont pas addressés individuellement aux *threads*.

Au moment de l'écriture de ce document[10] aucune implantation de MPI n'intégrait complètement le standard MPI-2.0 et l'on peut douter de le voir implantés un jour. Les plus grandes progrès se situent au niveau MPI-IO où nous trouvons différentes implantations : HPSS [104], PMPIO [70], MPI-IO/PIOFS [155] et ROMIO [156].

3.2.4 Bilan de l'état actuel

On peut établir un bilan d'une part sur le standard et d'autre part sur les implantation existantes.

Standard : La lacune fondamentale du standard est qu'il ne définit qu'une interface et pas un protocole de communication. Rien ne définit les formats de message, les en-têtes et les protocoles d'échange et de contrôle de flux. En particulier aucun protocole de routage ne permet d'utiliser plusieurs réseaux physiques. Il faudra toujours reposer sur TCP/IP pour cela. C'est à dire sur des implantations assez lourdes incapables d'exploiter des réseaux haut débits, ou alors sur versions particulières de MPI modifiées de façon spécifique. Si l'on accepte cette lacune, les fonctionnalités proposées par MPI-2 sont très complètes (et ambitieuses). On regrette, toutefois

[9] L'accès à MPI est fait en exclusion mutuelle

[10] Septembre, 1999.

l'absence des messages actifs et de la création de *threads* à distance.

Implantation : Le bilan est plus négatif du coté des implantations. Les implantations actuelles n'implantent que la version 1 du standard parfois étendue par MPIO. Ceci implique une compatibilité incertaine ou inefficace avec la multiprogrammation légère. De plus ces implantations ont choisi d'exploiter l'absence de définition d'un protocole pour optimiser les communications vers tel ou tel réseau. C'est particulièrement le cas des implantations propriétaires. C'est moins vrai pour les implantations « libres de droit » comme MPICH ou LAM. Par exemple, l'implantation MPICH peut utiliser le MPL (la bibliothèque de communication native du *HPS - High Performance Switch*) des machines IBM-SP comme mécanisme de base pour obtenir de faibles latences et un débit élevé, TCP/IP pour des réseaux de stations de travail standard (UNIX). Il est cependant impossible de les faire coopérer entre elles pour exploiter une grappe hétérogène au niveau réseau.

Le manque d'interopérabilité entre les différentes versions de MPI ou d'une même version sur différents réseaux sous-jacents, est un facteur gênant est bloquant pour le développement de projets de *metacomputing* [146] où plusieurs machines différentes sont interconnectées par des différents réseaux. Cela a été notre cas lors d'une expérience de connexion d'un T3E, d'un IBM SP1 et d'une grappe myrinet de PCs reliés par un réseau métropolitain ATM. Il n'a pas été possible de trouver une combinaison de bibliothèques MPI permettant d'acheminer les messages par le réseau à capacité d'adressage du T3E, par le réseau à haut débit du SP, et par le réseau myrinet.

Même l'utilisation d'une bibliothèque MPI (MPI LAM) tournant sur TCP/IP n'a pas été possible, ce protocole n'étant pas disponible sur le réseau rapide du T3E. Plusieurs projets de recherche sont en cours afin de proposer une solution au problème d'interopérabilité sans perdre l'efficacité offerte par des MPI « réglés » pour certaines architectures de système. Parmi ces projets, nous citons MPI-GLUE [133], PACX-MPI [80], PVMPI [64], MPIConnect [65], et I-WAY (Globus) [54].

Il n'est pas certain que le standard MPI-2 soit complètement implanté un jour. Cependant malgré ces lacunes, le standard MPI et ses implantations sont des outils utilisés de façon systématique sur les grappes de stations ou les machines parallèles. Cette utilisation importante provient d'une part de sa fiabilité mais aussi de l'existence d'indicateurs de performance pouvant guider le programmeur.

3.3 Le recouvrement calcul communication

Une méthode de parallélisation intuitive en calcul scientifique (parallélisme de données) consiste à découper un algorithme en alternance de phases de calculs et de phase d'échange de données entre ces calculs. Bien que facile à réaliser et à comprendre, ce style de programmation présente des problèmes de performance à cause de la sérialisation de ces deux phases [78]. Ceci peut être démontré par une

simple modélisation.

Si l'on suppose qu'à partir d'un programme séquentiel $P_{séq}$, exécuté en t_{seq} unités de temps, nous obtenons un programme parallèle P_{par} exécuté par p processus, de telle façon que le calcul est partagé équitablement entre ces processus (t_{seq}/p), nous pouvons estimer que le temps d'exécution parallèle t_{par} est donné par :

$$t_{par} = t_{calc} + t_{comm} = \frac{t_{seq}}{p} + t_{comm} \tag{3.1}$$

où t_{comm} représente le temps nécessaire aux différents processus pour recevoir les données à traiter et envoyer les résultats. L'accélération (*speedup*) obtenu par la parallélisation peut être exprimée comme :

$$S_p = \frac{t_{seq}}{t_{par}} = \frac{t_{seq}}{\frac{t_{seq}}{p} + t_{comm}} = \frac{p}{1 + \frac{t_{comm}}{t_{calc}}} \tag{3.2}$$

La **granularité** (γ) d'une application parallèle étant définie comme la relation entre le temps de calcul t_{calc} et le temps de communication (t_{comm}), nous permet de d'écrire l'équation 3.2 comme :

$$S_p = \frac{p}{1 + \frac{1}{\gamma}} \tag{3.3}$$

L'équation 3.3 nous montre que pour avoir une bonne accélération soit on augmente le « poids » de calcul par rapport aux communications, soit on diminue le temps de communication. La première solution nécessite un changement d'algorithme parallèle. Il en est de même pour la seconde car il faut réduire le volume de communication. Une autre solution est d'améliorer les opérations de communications (nous y reviendrons en 4.1 « La réalisation des communications »). En réalité il existe une troisième solution quand la contrainte stricte d'enchaînement calcul puis communication peut être relâchée. C'est le cas quand les communications ne conditionnent pas directement le calcul suivant (par exemple une émission). Il n'est pas nécessaire d'attendre sa terminaison. La méthode proposée réduit le temps apparent de communication en calculent en parallèle des communications ne conditionnant pas le calcul. On dit alors qu'on fait du « recouvrement calcul-communication ». On peut facilement voir que cela conduit à une amélioration de l'efficacité.

Le recouvrement calcul-communication considère qu'une partie des communication peut se faire en parallèle du calcul. Sous cette hypothèse, nous pouvons envisager les trois possibilités illustrées par la figure 3.2. Dans les cas (b) et (c) nous vérifions que le temps d'exécution de l'application avec le recouvrement ($t_{par_{rec}}$)

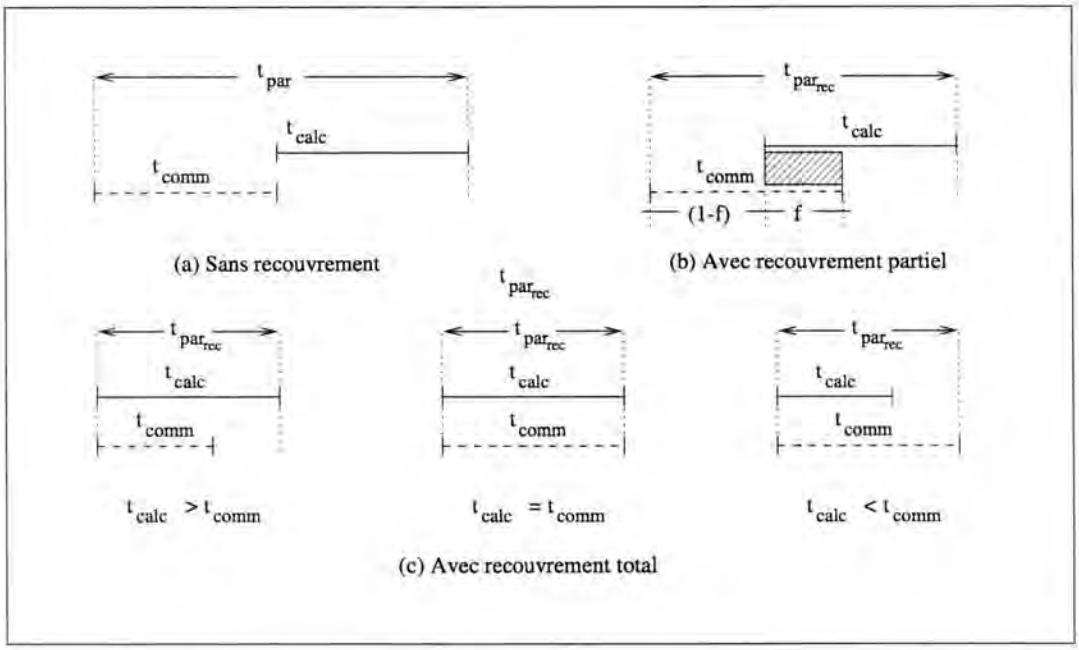

Figure 3.2 *Le principe du recouvrement calcul-communication*

est toujours inférieur au temps de l'exécution parallèle sans recouvrement (t_{par}). Le temps d'exécution parallèle est donné pour le cas (b) par :

$$t_{par_{rec}} = (1 - f)t_{comm} + max(ft_{comm}, t_{calc}) \tag{3.4}$$

où $f, 0 \leq f \leq 1$, répresente la fraction du temps de communication dont l'exécution est possible concurremment au calcul. Pour le cas (c), le temps d'exécution parallèle est donné par :

$$t_{par_{rec}} = max(t_{calc}, t_{comm}) \tag{3.5}$$

Le cas (c) montre une situation idéale. Il est utile pour nous donner une borne supérieure. A partir des équations 3.4, 3.5, et en considérant que la granularité γ est donnée par $\gamma = \frac{t_{calc}}{t_{comm}}$ nous pouvons, de manière similaire à l'équation 3.2, exprimer les accélérations pour ces deux cas comme :

$$S_{p_{rec}} = \frac{p}{\frac{(1-f)}{\gamma} + max(\frac{f}{\gamma}, 1)} \quad \text{(cas b)} \tag{3.6}$$

$$S_{p_{rec}} = \frac{t_{seq}}{max(1, \frac{1}{\gamma})} \quad \text{(cas c)} \tag{3.7}$$

La figure 3.3.a illustre l'effet d'un recouvrement f (équation 3.6) sur l'accélération, avec la borne inférieure donnée par l'équation 3.3 et la borne supérieure par l'équation 3.7. Une autre façon d'analyser cet effet est le gain obtenu par le recouvrement. Le gain est calculé comme la rapport entre le temps parallèle sans aucun recouvrement et le temps parallèle avec recouvrement f (fig. 3.3.b).

Sur les graphiques suivants, nous pouvons constater que le recouvrement est avantageux quand la granularité γ est petite. Quand $\gamma \rightarrow \infty$ le temps de calcul prédomine sur le temps total, et le gain apporté par le recouvrement tend vers zéro. Ceci justifie l'intérêt du recouvrement pour les applications à grain fin comme c'est la cas pour des applications irrégulières. La question qui se pose maintenant est : comment est-t-il possible de réaliser le recouvrement ? C'est à cette question que nous essayons de répondre par la suite.

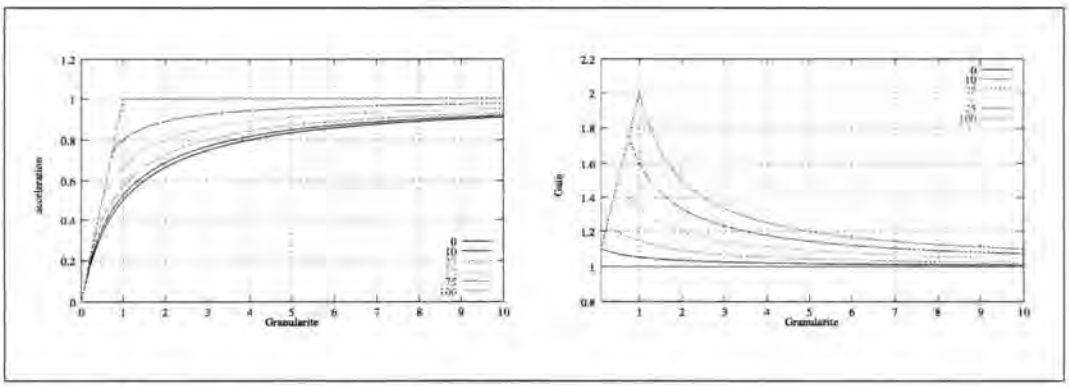

Figure 3.3 *L'influence du degré de recouvrement calcul-communication sur l'accélération*

3.4 Mise en œuvre du recouvrement

La mise en œuvre du recouvrement suppose que la communication puisse se faire en parallèle du calcul. Ceci suppose que les dispositifs d'émission, de réception et le médium de communication procédent de façon indépendante des processeurs. C'est en général le cas, et il est donc possible de recouvrir une partie du transfert effectif des données. Ce n'est pas le cas pour l'amorçage des transferts. Ceci veut dire que le recouvrement total des communications est soumis aux contraintes d'efficacité imposées par les mécanismes de détection et initialisation des transferts présentés au chapitre 4 (section 4.1). Du point de vue de la programmation, le recouvrement calcul-communication peut être réalisé de deux façons :

- par la programmation explicite de l'enchaînement des calculs et communications (via des opérateurs non bloquantes) de façon à ce que le système puisse acheminer les messages simultanément au calcul. Ceci suppose une indépen-

dance entre le périphérique de transfert de données et l'unité de calcul, ce qui est généralement le cas.

- par la découpe des calculs et communications en parties indépendantes qui sont affectés à des processus (ou *threads*) laissant ainsi la multiprogrammation assurer ces recouvrements. On suppose ici que lorsqu'un processus est bloqué sur une communication, un autre processus réalise des calculs.

Les choix de l'une ou de l'autre méthode ont une incidence sur les opérateurs de communication nécessaires et sur la « qualité » du recouvrement.

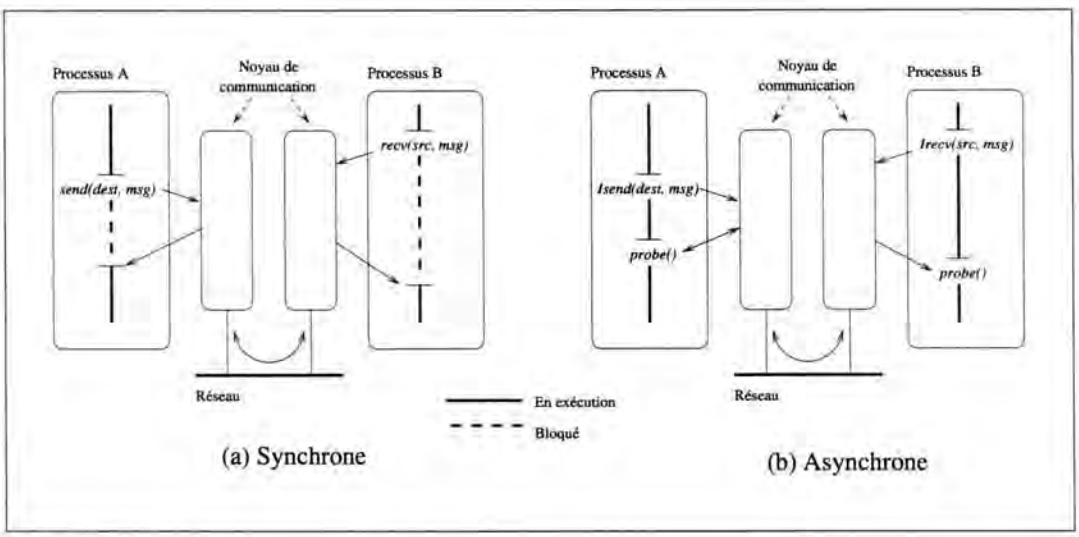

Figure 3.4 *La programmation asynchrone et le recouvrement calcul communication*

3.4.1 Communication asynchrone

Comme on l'a vu dans les sections précédentes à propos des bibliothèques de communication, les opérateurs de communications peuvent être bloquants (synchrone) ou non bloquant (asynchrone). Une tâche, lorsqu'elle effectue une primitive de communication synchrone, reste bloquée jusqu'au moment où les données sont envoyées (primitive *send*) ou que les données sont arrivées (primitive *recv*). La communication synchrone ne permet aucun type de recouvrement de calcul-communication (fig. 3.4.a).

En communication asynchrone, on n'utilise que des opérateurs non bloquants. Toute communication élémentaire se traduit en une demande de communication (requête) suivie ultérieurement par une primitive de test de terminaison (fig. 3.4.b). Le programmeur insère du calcul entre ces deux opérations pour obtenir le recouvrement calcul-communication. Cette méthode de programmation amène à des programmes moins structurés et plus sensibles à des erreurs, et par conséquent, plus difficile à écrire et à déboguer.

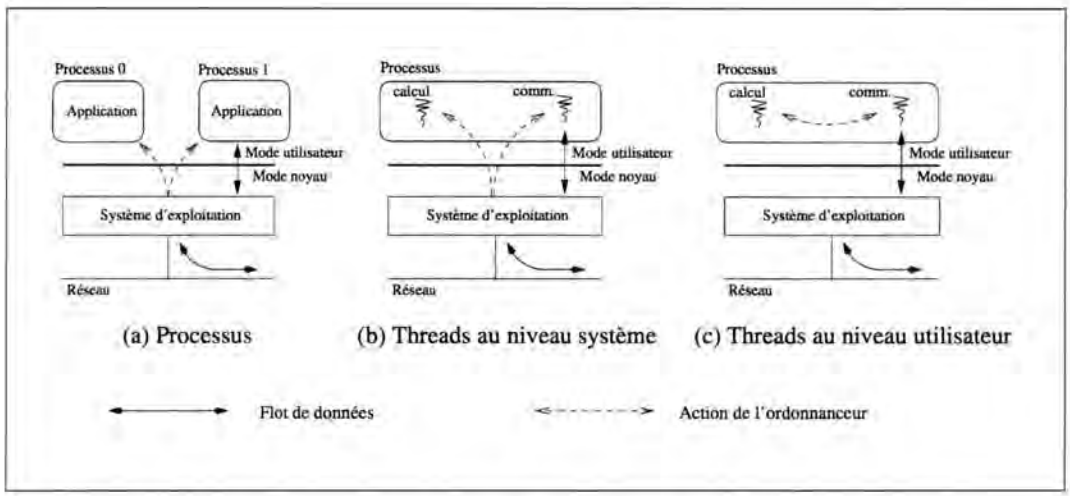

Figure 3.5 *Différentes supports pour le recouvrement calcul-communication*

3.4.2 L'emploi de la multiprogrammation légère

Dans ce cas, le programmeur organise son programme en divers processus regroupant les enchaînements de calcul et communications élémentaires. Les communications sont toutes synchrones. Le recouvrement est assuré par la multiprogrammation qui bloque le processus initiateur d'une communication, initie celle-ci et transfère le contrôle à un autre processus (fig.3.5.a). Ce principe est aisément adapté à la multiprogrammation légère.

On mesure l'intérêt d'utiliser la multiprogrammation légère pour obtenir un meilleur contrôle du grain de calcul et des communications. Une application parallèle est donc organisée en plusieurs *threads* à l'intérieur d'un même processus virtuel. Lorsqu'un *thread* commence une communication, il est bloqué. L'ordonnanceur du noyau de *threads* en choisit un autre pour continuer à calculer (fig.3.5, b et c). Le recouvrement calcul-communication est donc atteint. La figure 3.6 décrit schématiquement cette solution.

Cette solution présente le grand avantage de permettre une expression aisée du recouvrement, ce que donne des programmes mieux organisés, donc plus simples à comprendre, à développer et à déboguer. En plus du recouvrement obtenu par la multiprogrammation, il est toujours possible d'utiliser l'emploi des communications asynchrones. Ces avantages militent pour l'intégration des *threads* et de la communication. Le chapitre 4 est entièrement dédié à ce sujet. Auparavant, nous allons présenter les indicateurs de performance de la communication et leur utilisation dans le cas de MPI.

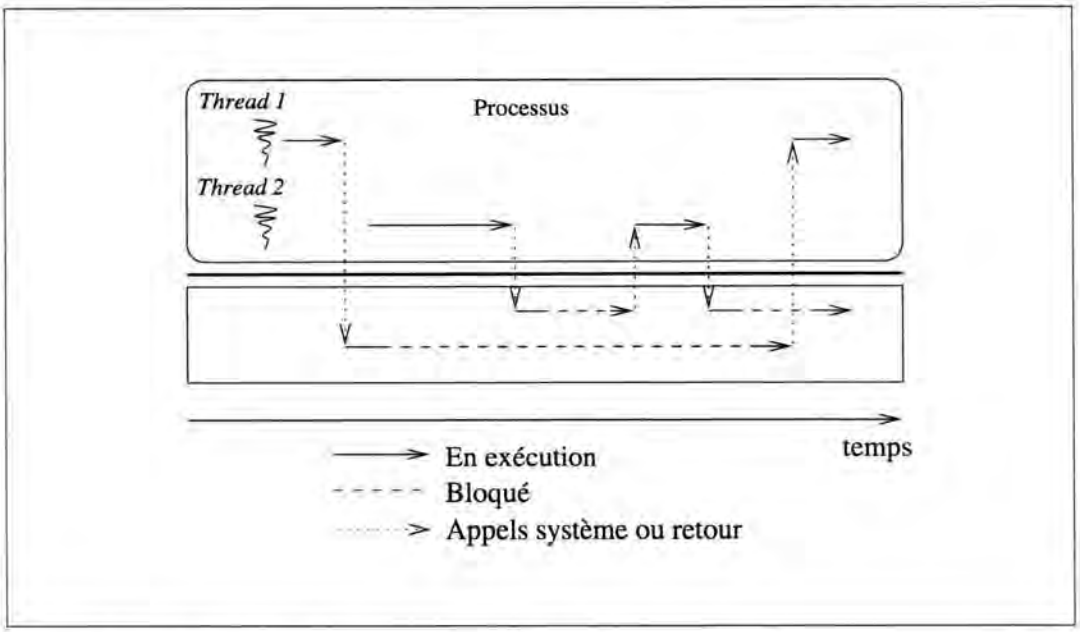

Figure 3.6 *Les* threads *et le recouvrement calcul communication*

3.5 Analyse de performance

Dans un soucis d'éviter la prolifération de jeux de tests[11] et d'indicateurs, le comité PARKBENCH (PARallel Kernels and BENCHmarks) a été crée lors du Supercomputing'92 à Mineapolis, USA, pour fournir un ensemble compréhensible acceptable par les constructeurs et les utilisateurs. La publication du standard PARKBENCH [46] présente des méthodes et jeux de tests permettant d'évaluer plusieurs aspects d'un système parallèle comme puissance de calcul, efficacité de communication, aspects liés au noyau système et aux compilateurs. Nous nous sommes plus particulièrement intéressés aux aspects portant sur l'évaluation de l'efficacité de la communication.

En effet l 'efficacité d'une application parallèle basée sur le paradigme d'échange de messages est fortement associée à celle de la bibliothèque de communication utilisée. Le standard MPI a été soigneusement conçu pour allier portabilité et efficacité. Cependant les implantions de MPI exploitent de façon diverses les caractéristiques des machines qui offrent elle mêmes des dispositifs de communication spécifiques. Il est donc intéressant de pouvoir évaluer l'efficacité d'une implantation MPI pour une machine donnée.

L'objectif de cette section est donc de décrire les indicateurs caractéristiques du comportement de la communication et les procédures d'évaluation de ces indicateurs.

[11] Benchmark.

3.5.1 Modèle de coût

La performance d'un réseau à niveau physique est caractérisée par deux indicateurs fondamentaux : **largeur de bande** (*bandwidth*) et **délai de transit** (*latency*). La largeur de bande est la quantité de bits qui peut être transmise dans une période fixe de temps. Cette métrique est exprimée en bits par seconde (bps). Le délai de transit correspond au temps nécessaire pour qu'un bit se propage d'un émetteur à un récepteur connecté au réseau. La bande passante est une caractéristique de la technologie utilisée pour la construction du réseau. Elle est directement liée à la fréquence de modulation, au codage des symboles, à la fréquence des resynchronisations et à la redondance nécessaire pour détecter ou corriger des erreurs. Le délai de transit est lié à la distance physique que le bit doit franchir c'est à dire à la longueur du medium (fil) et au nombre d'éléments commutateur à franchir. Ces performances sont des constraintes physiques d'un réseau et constituent des bornes supérieures de performance.

En effet, la gestion des communications comme le contrôle de l'interface introduisent des surcoûts. Le principal surcoût est celui occasionné par le système d'exploitation car il est de sa responsabilité d'implanter des protocoles de communications pour garantir la fiabilité d'une communication, pour réaliser l'emballage et le déballage des données, etc. Pour mesurer l'efficacité d'une bibliothèque de communication, nous avons donc besoin de tenir compte de ces surcoût c'est à dire de mesurer l'efficacité d'un canal logique de processus à processus au niveau applicatif.

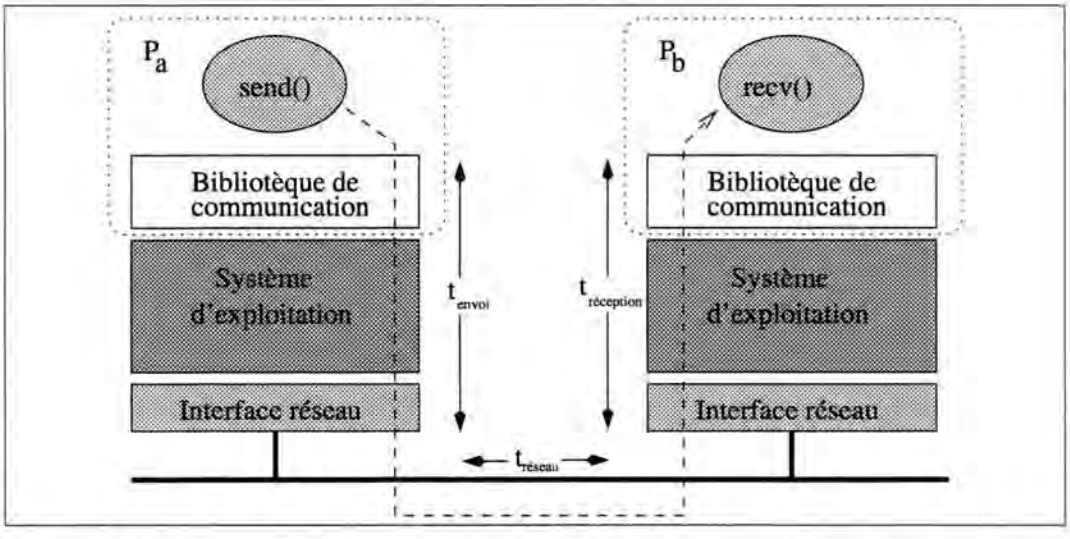

Figure 3.7 *Architecture générale d'une bibliothèque de communication*

La figure 3.7 présente l'architecture générale d'une bibliothèque d'échange de messages. Lors de l'envoi d'un message du processus P_A vers le processus P_B nous pouvons déterminer trois étapes prenant un temps significatif[124] :

– Le **temps d'envoi** représente le temps nécessaire pour préparer un message

et le délivrer au niveau réseau ;

- Le **temps réseau :** le temps pendant lequel le message circule sur le réseau pour aller de l'interface émettrice jusqu'à l'interface réceptrice ;
- Le **temps de réception :** associé au temps écoulé entre l'arrivée du message à l'interface réseau et sa livraison au processus destinataire en supposant celui-ci disponible (temps minimum) ;

Par analogie au réseau physique, on mesure les deux indicateurs de performances : le **délai de bout-en-bout** (*communication latency*) et le **débit** (*throughput*). Le *délai de bout-en-bout* est l'intervalle de temps mesuré entre le moment où l'émetteur demande à émettre et celui où le récepteur reçoit le message. Si l'on suppose que la réception est toujours prête et que la charge du réseau est nulle, nous pouvons écrire le *délai de bout-en-bout* (t) comme :

$$t = t_s + n \times t_t + \lfloor n/p \rfloor \times t_p \tag{3.8}$$

où t_s représente le coût **fixe** introduit par les appels aux fonctions nécessaires pour l'envoi (ou la réception) d'un message indépendamment de sa taille, n la taille du message en octects, t_t le temps de transmission physique d'un octet, et t_p le temps nécessaire pour emettre/recevoir le message logique en unités de transmission et réception (p). En effet, un réseau transmet généralement les données par paquets de taille fixe ou bornée. Ce temps de transfert est modelisé par une fonction croissante avec la taille du message.

Le deuxième indicateur, le *débit* (δ), est le taux de transfert maximal (en octets par seconde) que la bibliothèque de communication est capable d'offrir :

$$\delta = \frac{\text{Taille du message}}{\text{Temps du transfert}} \tag{3.9}$$

où le temps de transfert prend en compte la largeur de bande offerte par le réseau, les surcoûts introduits par les couches système et par la bibliothèque de communication. En effet, le temps de transfert est le *délai bout-en-bout* défini par l'équation (3.8) ce que nous permet d'écrire (3.9) comme :

$$\delta = \frac{n}{t_s + n \times t_t + \lfloor \frac{n}{p} \rfloor \times t_p} \tag{3.10}$$

On considère que la charge du réseau est nulle. Selon la taille du message, le coût d'amorçage d'une communication (t_s) et son coût d'entretient ($t_p \times \lfloor \frac{n}{p} \rfloor$) sont plus au moins importants relativement au coût effectif de transfert. Selon le cas, un programmeur pourra ou non regrouper des petits messages en de plus gros.

3.5.2 Indicateurs de performance

Hockney[95] propose de rendre compte de l'effet des surcoûts pour un réseau donné par des indicateurs calculables à partir de mesures simples. Ces indicateurs sont appéles $r_\infty, n_{\frac{1}{2}}, \pi_0$ et t_0. Ces indicateurs ont été conçus à l'origine pour évaluer la performance des unités arithmétiques des ordinateurs vectoriels. Cependant Hockney démontre que cette caractérisation est valable pour n'importe quel système présentant un comportement linéaire par rapport à une variable. A la section précédente nous avons vu qu'un modèle de coût simple pour une bibliothèque de communication est linéaire c'est à dire que le temps de transfert est donné par une fonction de type $t = \alpha + \beta \times n$ où n est la taille du message.

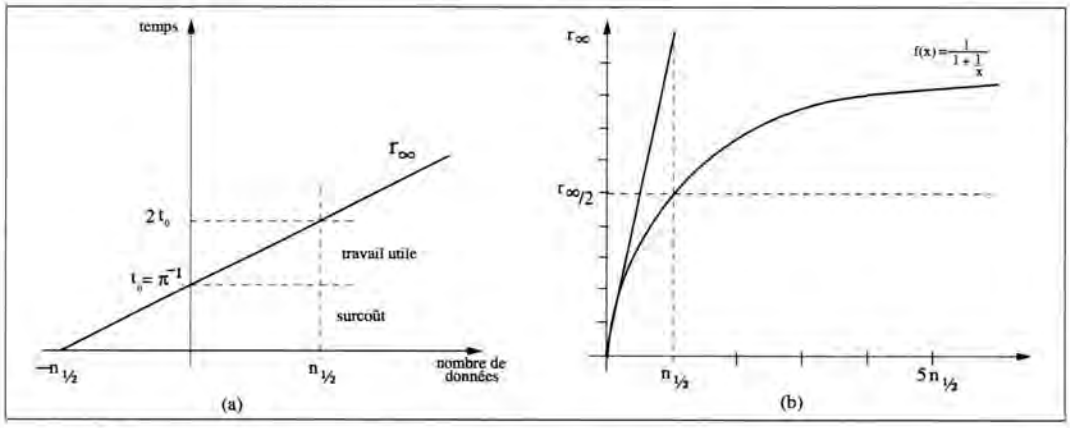

Figure 3.8 *Interprétation des paramétres* $(r_\infty, n_{\frac{1}{2}}), (\pi_0, n_{\frac{1}{2}}), (t_0, r_\infty)$

La figure 3.8 donne la représentation graphique de ces paramètres : l'indicateur r_∞ correspond à l'inverse de la pente de la droite $(1/\beta)$, et $n_{\frac{1}{2}}$ le point de la fonction où l'ordonnée a une valeur équivalente à deux fois le terme constant (α) de l'équation de la droite. Nous allons expliquer le rôle ces indicateurs dans la caractérisation d'un réseau (communications) :

- r_∞ est le débit de communication maximal offert par le réseau.
- $n_{\frac{1}{2}}$ est la taille du message présentant une durée de transfert égale au coût d'amorçage (α). $n_{\frac{1}{2}}$ est dit taille de message à mi-débit car il représente une utilisation du réseau égale à 50% de r_∞.

Donc la relation $n_{\frac{1}{2}}/r_\infty$ nous fournit l'importance du coût initial de l'opération de transfert par rapport au nombre de données transférées. On peut d'ailleurs reécrire l'équation de durée de transfert par :

$$t = r_\infty^{-1} \times (n_{\frac{1}{2}} + n) \tag{3.11}$$

À partir de 3.9 et 3.11 il est possible de définir le débit de transfert effectif (r) comme le rapport entre le nombre de données transférées et le temps de transfert :

$$r = \frac{\text{Taille du message}}{\text{Temps du tranfert}} = r_\infty \times \frac{1}{1 + \frac{n_{\frac{1}{2}}}{n}} \tag{3.12}$$

Le débit effectif r est donné par une fonction de la forme $f(x) = 1/(1 + 1/x)$ qui tend asymptotiquement vers une limite pour x croissant. Pour des messages de grande taille ($n \gg n_{\frac{1}{2}}$), r tend vers r_∞ qui est le débit de communication maximal offert par le réseau.

Pour les messages de petite taille ($n \ll n_{\frac{1}{2}}$), le débit de communication est approximativement proportionnel à la taille du message. On appelle cette proportion **débit spécifique** noté (π_0). Sa valeur est donnée par $r_\infty/n_{\frac{1}{2}}$. Elle permet d'estimer rapidement le débit qui l'on peut atteindre pour une petite taille de message simplement en faisant le produit de l'indicateur π_0 par la taille du message. Enfin l'inverse de π_0 ($n_{\frac{1}{2}}/r_\infty$) est l'indicateur t_0 donnant le temps d'amorçage (soit α) de toute émission et aussi la durée d'émission effective du message à mi-débit).

En général nous pouvons considérer que l'indicateur r_∞ caractérise la performance limite pour de messages de grande taille et π_0 pour ceux de petite taille. L'évaluation de tous les indicateurs $(r_\infty, n_{\frac{1}{2}}), (\pi_0, n_{\frac{1}{2}}), (t_0, r_\infty)$ est nécessaire pour caractériser l'efficacité de la communication offerte par une bibliothèque MPI donnée sur un réseau donné. Il est à la charge du programmeur d'exploiter ces propriétés en jouant sur la taille des messages, le débit de requêtes, etc. En particulier, il est important de regarder plus précisement la courbe mesurée du débit effectif (qui est toujours de la forme de la fig. 3.8.b). Ainsi si la courbe mesurée suit ce tracé on déduit que 80% du débit optimal est atteint pour un message de taille égale à $4 \times n_{\frac{1}{2}}$, 90% pour des message de taille $9 \times n_{\frac{1}{2}}$. Cette information est utile au programmeur pour la définition de la taille du message à utiliser afin de maximiser un rapport coût/performance.

3.5.3 Suites PARKBENCH

Le standard PARKBENCH, dans la suite *« low-level »*, ou COMMS, offre un ensemble de jeux de tests (COMMS1, COMMS2, et COMMS3) pour mesurer les indicateurs $(r_\infty, n_{\frac{1}{2}}), (\pi_0, n_{\frac{1}{2}})$ et (t_0, r_∞) d'une bibliothèque de communication sous différents régimes d'opération. Une autre suite intéressante est la suite *« balance benchmarks »*, ou POLY, utilisée pour estimer le rapport entre le temps de calcul et le temps nécessaire pour transférer les données impliquées. Nous présentons ces tests dans les paragraphe suivants.

La suite COMMS : Le test ping-pong

Le test COMMS1, aussi connu sous le nom de *ping-pong* ou *echo*, mesure le délai de bout-en-bout nécessaire pour envoyer un message d'un processus A à un

processus B. Généralement il n'est pas possible d'obtenir directement le délai de bout-en-bout sur un réseau de stations de travail car les stations n'ont pas d'horloges synchronisées. Une façon de contourner ce problème est de diviser par deux le temps mis par un message pour effectuer un aller-retour[12] entre ces deux processus[13]. De cette manière il est possible de mesurer le temps sur une seule machine. Le test de *ping-pong* consiste donc à envoyer un message d'une taille donnée d'un processus A (serveur) vers un processus B (client). Quand le client reçoit le message il le renvoie vers le serveur. La variation de la taille du message envoyé permet d'évaluer l'incidence de la taille du message sur le délai.

Une proprieté importante d'une bibliothèque de communication est d'atteindre le débit optimal non seulement pour un seul gros message mais pour plusieurs messages de plusieurs processus. Les tests suivants sont destinés à évaluer le comportement de la bibliothèque en charge.

Le test COMMS2 est une variation du test *ping-pong* où une paire de processus s'envoient des messages l'un vers l'autre simultanément. Ce test permet d'analyser la capacité de la bibliothèque à gérer en même temps l'envoi et la réception de messages principalement si le réseau physique offre une capacité bidirectionnelle. La généralisation de la procédure COMMS2 est d'imaginer un système avec p processus où chaque processus i envoie un message de taille n aux autres $(p-1)$ processus $j, j \neq i$. L'objectif est de saturer le débit de la bibliothèque de communication et de vérifier comment son comportement varie avec l'augmentation du nombre de processus. Cette généralisation constitue le test COMMS3.

La suite POLY

La suite POLY a été initialement conçue pour évaluer le rapport entre le taux de transfert des données de la mémoire vers l'unité arithmétique d'un ordinateur vectoriel. En général de telles unités reçoivent deux arguments en entrée et produisent un résultat. Ainsi pour que la mémoire alimente en continu l'unité de calcul, le débit de transfert mémoire (mesuré en Mo/s) doit être trois fois (2 lectures et 1 écriture) plus rapide que le débit de calcul (mesuré en Mflops/s). Si le débit mémoire ne suit pas celui de l'unité de calcul, les accès mémoire deviennent un goulot d'étranglement.

L'indicateur de performance proposé par Hockney et Curington [96] pour évaluer l'équilibre entre les accès mémoire et l'unité de calcul est l'**intensité computationelle** (f) laquelle est défine par le rapport du nombre total d'opérations de calcul (travail) à faire et du nombre d'accès aux données pour faire ce travail. Si l'intensité computationelle est grande cela signifie que le temps de transfert de la mémoire est faible par rapport le temps de calcul. Par ailleurs, si l'intensité computationelle est petite cela signifie que c'est le temps de transfert qui prédomine. Hockney montre que si l'intensité computationnele augmente, le délai dû aux accès mémoire devient

[12] Round Trip Time (RTT).

[13] on suppose que le temps d'envoi et le temps de recéption sont équivalents.

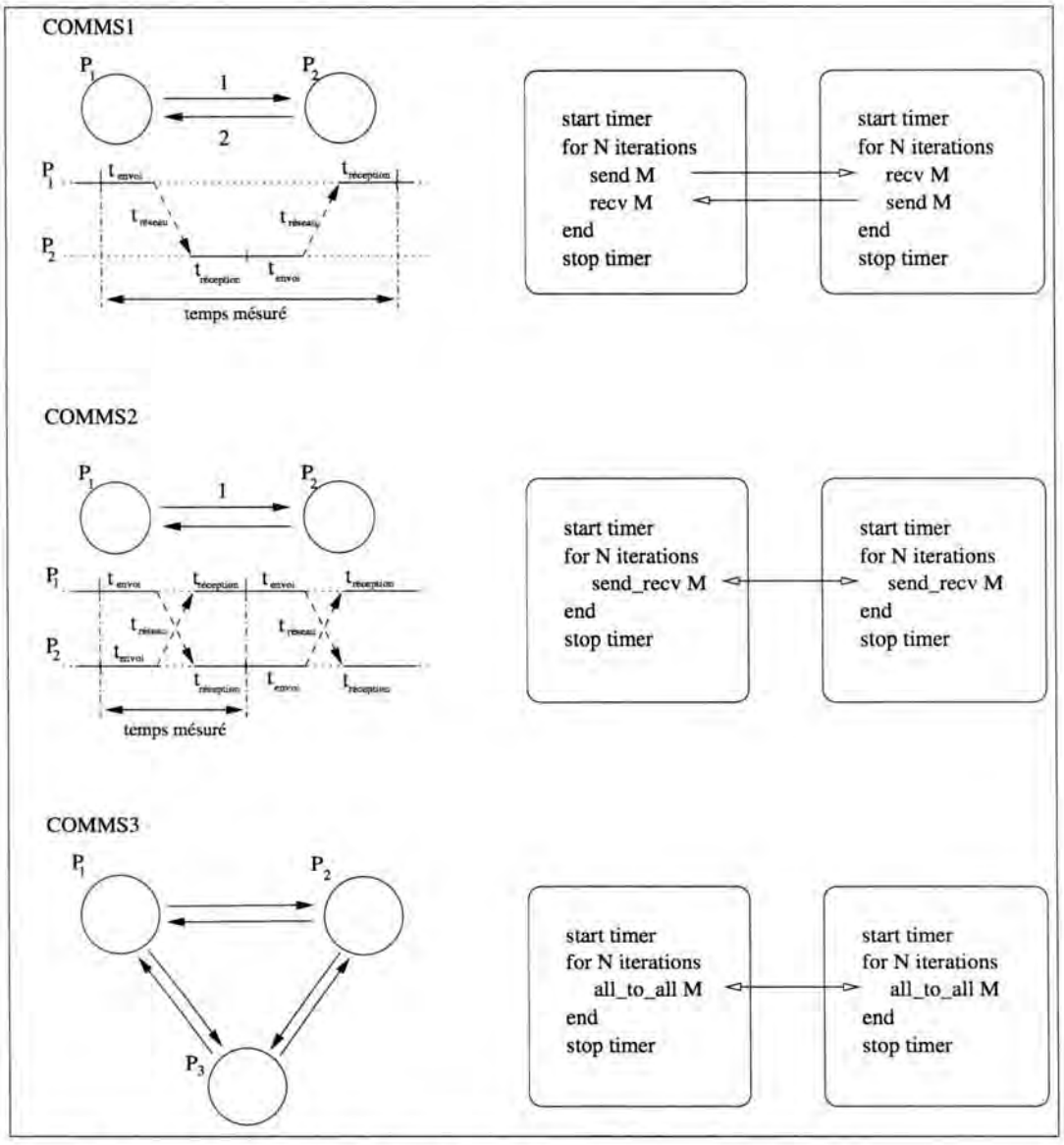

Figure 3.9 *La suite COMMS de PARKBENCH*

négligeable par rapport au temps passé dans le calcul, et sa valeur tend à une asymptote ($\hat{r}_\infty$). Par analogie avec l'indicateur $n_{\frac{1}{2}}$, la **demi-intensité computationelle**($f_{\frac{1}{2}}$) est défini comme l'intensité computationelle nécessaire pour atteindre la moitié de la valeur de l'asymptote.

L'implantation de la suite POLY est basée sur le calcul du polynome :

$$p(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + a_n x^n$$

par la règle de Horner :

$$p(x) = a_0 + x(a_1 + \cdots + x(a_{n-2} + x(a_{n-1} + a_n x)) \cdots)$$

Dans ce cas, l'évaluation d'un polynome $p(x)$ de degré n, nécessite d'effectuer $n + 4$ accès mémoire et $2n$ opérations arithmétiques [160]. Si l'on réalise le calcul de $p(x_i)$,$i = 0, 1, \ldots, k - 1$ pour l'ensemble $X = x_0, x_1, \ldots, x_{k-1}$, la cardinalité de X permet l'analyse de différentes situations.

La suite POLY évalue l'intensité computationnelle f pour trois cas : les données sont dans le cache (POLY1), les données sont dans la mémoire (POLY2), et les données sont placées sur un autre processus (POLY3). Nous nous sommes intéressés au cas POLY3. Ici l'intensité computationnelle est interpretée comme le rapport entre le nombre d'opérations à faire et le temps de transfert des données nécessaires au travail. L'algorithme utilisé par POLY 3 est schématisé par la figure 3.10.

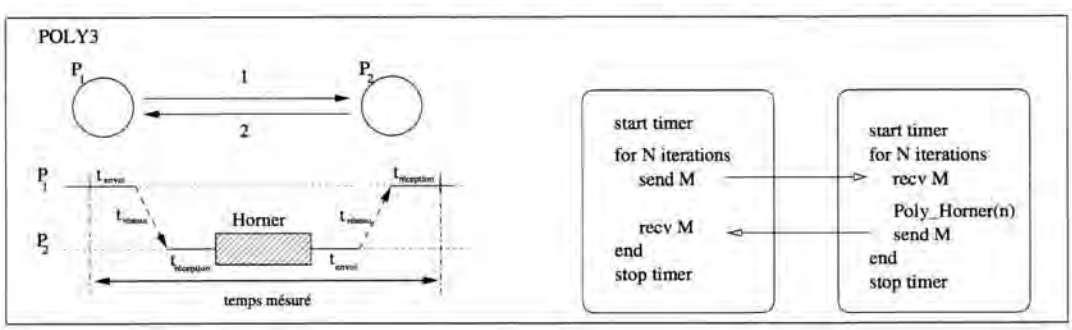

Figure 3.10 *Le* benchmark *POLY3 de la suite PARKBENCH*

L'indicateur d'intensité computationnelle doit aider les développeurs de programmes parallèles à définir une découpe en processus limitant le surcoût dû aux communications pour qu'il ne devienne pas trop important. Puisque l'intensité computationnelle tend vers une asymptote, le résultat de la section 3.5.2 est encore valable, c'est à dire 80% de la valeur de l'asymptote est obtenue par un rapport calcul-communication équivalent à $4 \times f_{\frac{1}{2}}$.

3.5.4 Le recouvrement calcul-communication par communication asynchrone

Nous avons vu à la section 3.4.1 qu'on peut en utilisant la communication asynchrone superposer la réalisation de la communication par des calculs. On gagne donc le temps de cette superposition. On peut se poser les questions suivantes : ce temps est-t-il considérable ? Quel est son ordre de grandeur ? Justifie-t-il une difficulté additionnelle de programmation pour qu'il soit exploité ? Pour estimer ce temps nous avons développé un test basé sur l'exécution d'un programme synthétique qui présente une structure similaire à celle qu'on retrouve sur certaines applications parallèles comme Jacobi [17]. Cet algorithme suppose la division du

domaine de calcul (un plan) en plusieurs sous domaines (en blocs, en lignes, ou en colonnes) ; pour chaque point d'un sous domaine on réalise une « moyenne » en considèrent les valeurs des points immédiatement à gauche, à droite, en haut et en bas. Les points sur la frontière d'un domaine doivent donc être échangés avec les domaines voisins. Les implantations de Jacobi utilisant la programmation par échange de messages sur grappes de stations suivent le schéma général donné par l'algorithme 3.1.

Algorithme 3.1 La structure de l'algorithme de Jacobi

1: Evaluer les points limites de l'itération i+1
2: Envoyer les points limites de l'itération i+1 aux domaines voisins
3: Evaluer Jacobi sur le domaine (itération i)
4: Lire les points limites des domaines voisins

On notera que l'algorithme 3.1 exploite l'envoi des points limitrophes de l'itération $i+1$ (pas 2) simultanément à l'itération i (pas 3). Pour simplifier notre discours à propos du recouvrement offert par ce type d'organisation de programme parallèle, nous allons définir un programme synthétique présentent la même structure. Nous allons donc réaliser le calcul du nombre π par l'évaluation de l'intégrale :

$$\pi = \int_0^1 \frac{4}{(1+x^2)} dx.$$

L'implantation du programme synthétique est similaire à un *ping-pong* à l'exception qu'on introduit le calcule de π par la méthode des trapèzes (fig. 3.11). Le « poids » de calcul entre l'intervalle d'intégration a_0 et c_0 est constant pour un nombre n de sous intervalles. Donc évaluer cette intégrale en utilisant deux intervalles, $[a_0; b_0]$ et $(b_0, c_0]$, représente le même « poids » de calcul. L'idée est de réaliser des envois de messages d'un processus à l'autre, en partageant l'évaluation de π en deux intervalles : un entre la primitive *isend/test* et *irecv/test* ($calcul_1$), l'autre après le *irecv/test* ($calcul_2$)[14].

3.6 Évaluation de l'implantation MPI LAM 6.3

La bibliothèque MPI que nous avons choisie est LAM. Le choix de LAM est dû au fait que cette bibliothèque s'est montrée plus robuste en présence de *threads* que la bibliothèque MPICH sur des grappes interconnectées par un réseau TCP/IP. Le

[14]En principe, pour vérifier le recouvrement il suffit que le *send* avant le $calcul_1$ soit du type non bloquant. Cependant nous avons utilisé pour toutes les opérations de communication leurs versions asynchrones, suivis par des tests de terminaison. Ceci dans le but d'éviter des coûts additionnels de synchronisation entre les processus.

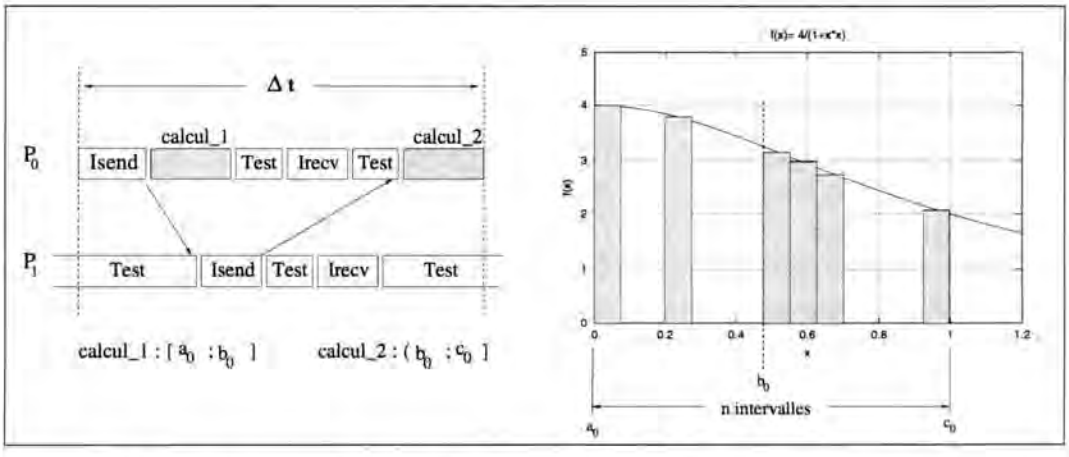

Figure 3.11 *L'évaluation de π par la méthode de trapèzes*

problème provient du fait que MPICH repose sur la bibliothèque P4 qui ne supporte pas de coexister avec des *threads*.

L'objectif de cette section est de mesurer les indicateurs de performance $r_\infty, n_{\frac{1}{2}}$ et t_0 pour LAM 6.3 et vérifier jusqu'à quel point ses opérateurs de communication de base (*send* et *recv*) rendent possible une exploitation du recouvrement calcul-communication. Ultérieurement nous utiliserons ces résultats pour évaluer la performance d'ATHAPASCAN-SMP (chapitre 6).

3.6.1 Les indicateurs $r_\infty, n_{\frac{1}{2}}$ et t_0 pour LAM 6.3

La bibliothèque de communication LAM est organisée en deux couches. La couche supérieure est la couche de portabilité qui offre les primitives MPI indépendemment du sous système de communication. La couche inférieure, appelée *Request Progression Interface* (RPI), implante les mécanismes nécessaires pour la réalisation des communications sur un interface spécifique. Elle offre ses fonctionnalités en deux versions. Une version considérée comme « portable », laquelle peut être employée sur n'importe quel système UNIX. Cette version utilise un démon de communication (fig. 3.1.a) pour réaliser les communications et offrir des facilités pour l'observation et débogage d'applications MPI.

Le deuxième version est appelée *client-to-client* (*c2c*) et elle est censée offrir une communication directe et efficace entre les processus MPI. Les fonctions qui la composent doivent être récrites à chaque portage de LAM sur un nouveau interface de communication pour mieux exploiter ses capacités. Actuellement le mode *c2c* présente 3 implantations : *tcp,sys* et *usysv*. Le node *tcp-c2c* utilise le protocole TCP/IP. Les modes *sys-c2c* et *usysv-c2c* emploient les mémoires partagées offertes par UNIX pour communiquer entre les processus MPI placés sur un même nœud physique et le protocole TCP/IP pour ceux qui sont placés sur des nœuds distincts. La différence entre ces deux modes est la façon dont ils réalisent le verrouillage

pour l'accès mémoire, l'un emploie des verrous et l'autre des *spin lock*[15].

L'implantation de LAM utilise encore deux protocoles différents pour acheminer les messages : *short* et *long*. Le protocole *short* est utilisé pour l'envoi de messages « petits ». Dans ce cas le message est envoyé en une seule opération de transfert. Le protocole *long* est réalisé en deux parties : l'envoi d'un enveloppe et après l'envoi du message (cf. section 3.1.2). LAM MPI, par défaut, définit pour le mode *c2c*, une taille de 64k octets comme point de changement de protocole. En mode démon, cette valeur est fixée à 8k octets.

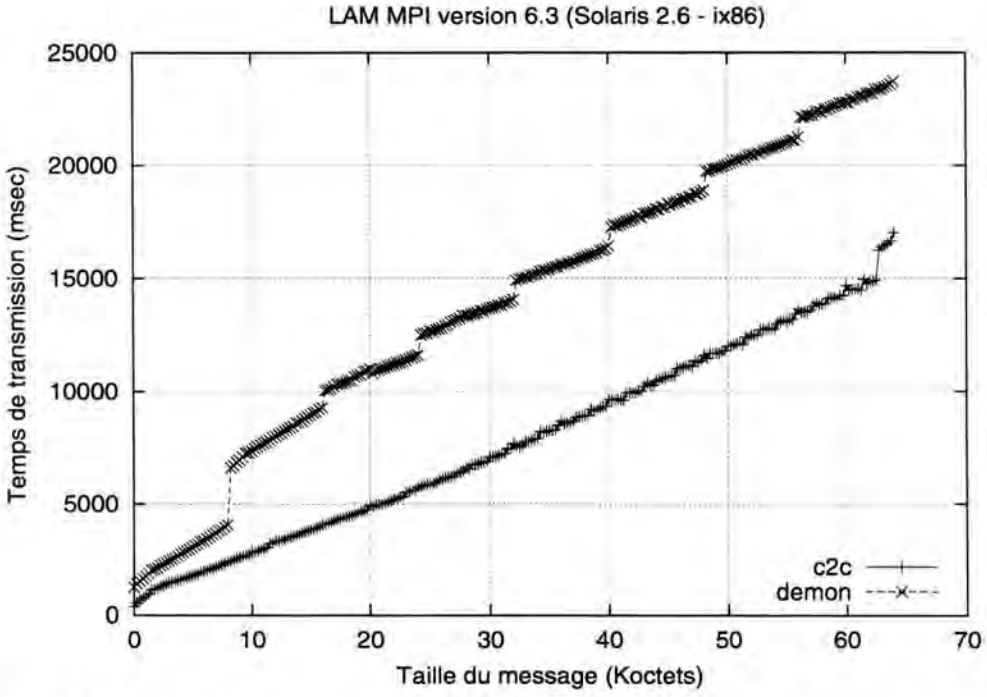

Figure 3.12 *Temps de transmission pour LAM MPI (version 6.3)*
(Plate-forme : 2 bi-pentium II 333 Mhz, solaris 2.6, Ethernet 100 Mbps)

Nous avons donc réalisé le *benchmark* COMMS1 pour caractériser le comportement de LAM sur nos machines (annexe B) à partir des indicateurs r_∞, $n_{\frac{1}{2}}$ et t_0. Ici nous ne présentons que les courbes obtenues par les biprocesseurs. Les monoprocesseurs, au niveau forme des courbes, donnent des résultats similaires. A partir des figures 3.12, 3.13, et 3.14 nous constatons que :

- Le surcoût imposé par le démon LAM est assez important.
- Le saut à 8k octets pour le mode démon correspond au changement de protocole *short/long*.
- Le protocole *long* pour le mode démon est implanté par une zone de mémoire partagée. Pour éviter qu'un gros message monopolise entièrement cette zone

[15]Ce type de verrou est présenté au chapitre 5.

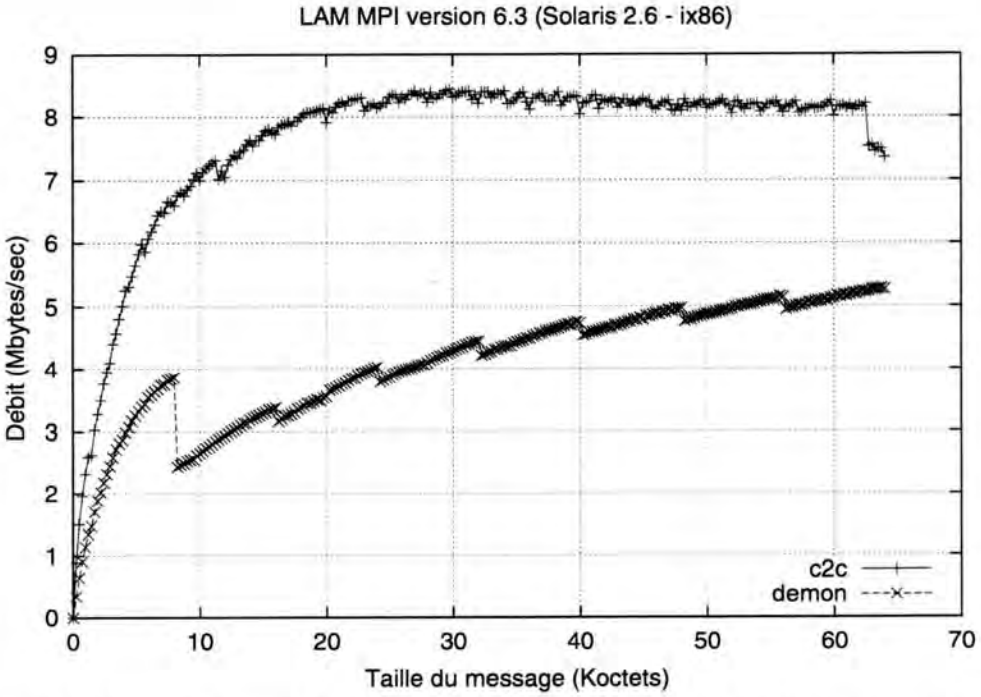

Figure 3.13 *Débit pour LAM MPI pour des messages jusqu'à 64 Koctets (version 6.3) (Plate-forme : 2 bi-pentium II 333 Mhz, solaris 2.6, Ethernet 100 Mbps)*

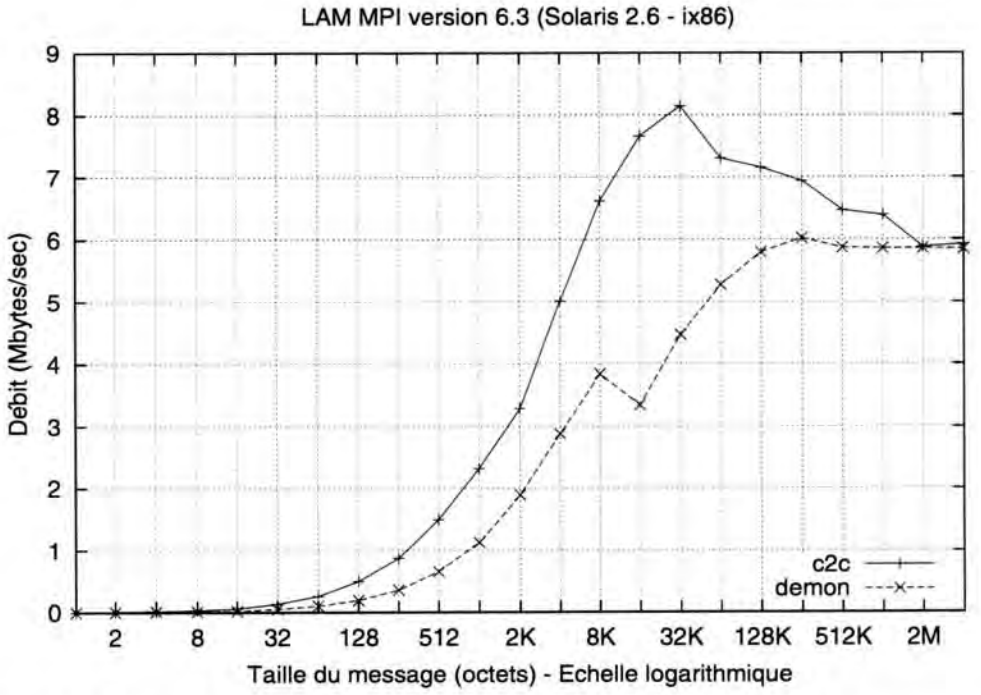

Figure 3.14 *Débit pour LAM MPI pour des gros messages (version 6.3) (Plate-forme : 2 bi-pentium II 333 Mhz, solaris 2.6, Ethernet 100 Mbps)*

LAM impose une gestion de tampons de 8k octets, ce qui explique les sauts intermédiaires à chaque multiple de 8k octets.

- Une dégradation de performance aux environs de 63k octets pour le mode *c2c*. On n'a pas pu déterminer la cause exacte. Apparemment, elle est due à la gestion de tampons de la pile TCP/IP (cette dégradation est aussi présente sur un *ping-pong* TCP basé sur *sockets*).
- On remarquera une différence de la pente entre le début d'envoi de messages et la suite des envois pour le mode *c2c*. Le point de changement est aux environs de 1.5k octets ce qui correspond au MTU[16] du protocole Ethernet. Les messages de taille supérieure au MTU provoquent un surcoût de fragmentation.

Etant donné les régimes de fonctionnement (*short/long*) nous avons décidé de déterminer l'indicateur r_∞, et par conséquent $n_{\frac{1}{2}}$, pour les deux cas. Ces valeurs sont présentées aux tableaux 3.1 et 3.2.

Tableau 3.1 *Les métriques r_∞,t_o ,$n_{\frac{1}{2}}$ pour le protocole* short *(LAM version 6.3 - Ethernet 100 Mbps)*
Les valeur entre parenthèses fournissent l'écart pour un intervalle de confiance égal à 90%

	Débit(r_∞^{64ko})	latence(t_o)	Mi-débit($n_{\frac{1}{2}}$)
Biproc. (c2c)	8.20 Mo/s	195μs($\pm$0.5)	$\approx$3.0 Ko
Biproc. (démon)	3.86 Mo/s	610μs($\pm$2.0)	$\approx$2.0 Ko
Monoproc. (c2c)	3.80 Mo/s	320μs($\pm$3.0)	$\approx$ 2.75 Ko
Monoproc. (démon)	1.38 Mo/s	2020μs($\pm$32.0)	$\approx$2.25 Ko

Tableau 3.2 *Les métriques r_∞,t_o ,$n_{\frac{1}{2}}$ pour le protocole* long *(LAM version 6.3 - Ethernet 100 Mbps)*
Les valeur entre parenthèses fournissent l'écart pour un intervalle de confiance égal à 90%

	Débit(r_∞^{4Mo})	latence(t_o)	Mi-débit($n_{\frac{1}{2}}$)
Biproc. (c2c)	5.9 Mo/s	195μs($\pm$0.5)	$\approx$1.75 Ko
Biproc. (démon)	5.8 Mo/s	610μs($\pm$2.0)	$\approx$ 4.0 Ko
Monoproc. (c2c)	3.3 Mo/s	320μs($\pm$3.0)	$\approx$ 2.0 Ko
Monoproc. (démon)	1.7 Mo/s	2020μs($\pm$32.0)	$\approx$3.25 Ko

3.6.2 Recouvrement calcul-communication pour LAM 6.3

Pour mesurer le recouvrement calcul-communication offerte par LAM 6.3 sur nos plate-formes nous avons utilisé la méthodologie décrite à la section 3.5.4. Fon-

[16]Maximum Tranfer Unit

damentalement on réalise un *ping-pong* où les opérations de *send* et *recv* sont séparées par un calcul dont le « poids » varie (fig. 3.11). Le résultat obtenu par l'exécution du programme synthétique de calcul de π est présenté à la figure 3.15. Cette figure fournit les temps de calcul obtenus lorsqu'on divise l'intervalle d'intégration en 6000 sous intervalles. Cela nous donne un « poids » total de calcul de $\approx 670\mu s$. Nous rappelons que le temps d'envoi d'un message de taille nulle est de $\approx 390\mu s$. On débute avec tout le « poids » de calcul sur la partie $calcul_2$ c'est à dire qu'on réalise un *ping-pong* avec une charge de calcul attachée à la fin du *recv*. Ensuite on commence à deplacer une partie du $calcul_2$ vers le $calcul_1$. L'abcisse nous donne le nombre d'itérations effectuées par $calcul_1$ par rapport au total (6000). Cette opération est répétée pour plusieurs tailles de messages. Toutes les courbes sont identiques. On atteint une durée d'exécution minimale. Appelons saturation, le nombre d'itérations de calcul pour lequel cette durée minimale est atteinte.

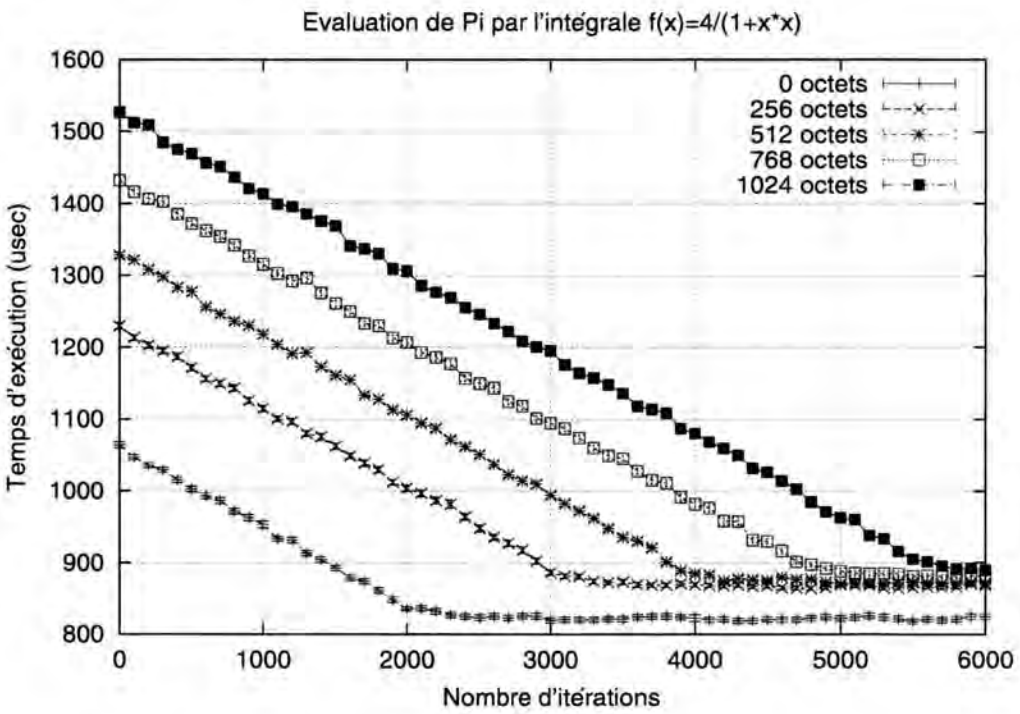

Figure 3.15 *Exemple de recouvrement calcul-communication : calcul de π (Plate-forme : 2 bi-pentium II 333 Mhz, solaris 2.6, Ethernet 100 Mbps)*

Analysons le cas des messages de taille nulle. Le temps total d'exécution du *ping-pong* suivi du calcul est de $\approx 1064\mu s$ qui correspond au temps d'un aller-retour d'un message de taille nulle ($\approx 390\mu s$) plus le temps de calcul ($\approx 670\mu s$). Au fur et à mesure que nous transférons une parcelle du $calcul_2$ vers le $calcul_1$ nous observons une diminution du temps de calcul. Ceci va arriver jusqu'à un point de saturation du temps de calcul à $\approx 820\mu s$, ce qui correspond à un gain de temps de calcul équivalent à $\approx 240\mu s$. Ce gain représente le gain apporté par le recouvrement

des communications par de calcul. A partir de la figure 3.15 on peut constater encore que :

- Tant que le temps de communication est supérieur au temps de calcul il est possible de recouvrir. Après, le temps de calcul prédomine et l'on ne gagne plus (limite inférieure du temps de calcul).
- Le point de flexion des courbes délimite à gauche le temps de communication qui peut être recouvert par des calculs. La valeur de $\approx 240\mu s$ représente le grain minimal possible de recouvrement.
- Les courbes sont décalées d'un facteur constante d'environ $100\mu s$ à chaque 256 octets. Pour le message de taille nulle, il doit y avoir une optimisation interne car il existe une différence de $\approx 40\mu s$ entre ce cas et l'envoi d'un message d'un octet. Cette valeur correspond à l'écart à saturation entre la courbe de zéro octets et la courbe de 256 octets.
- Il existe un seuil au dessous duquel on ne peut pas recouvrir. Cette limite correspond aux surcoûts de la bibliothèque de communication. Nous pouvons estimer ce surcoût par la différence entre le temps d'envoi d'un message d'un octet ($430\mu s$) et le temps de recouvrement maximal exploité par ce cas ($240\mu s$). Ceci nous donne $210\mu s$ comme surcoût. On peut vérifier cela en analysant le recouvrement obtenu par un autre message, par exemple, 768 octets. Ce message a un temps d'aller-retour équivalent à $\approx 760\mu s$. Le temps de communication plus le temps de calcul donne $\approx 1432\mu s$, on atteint une durée minimum de $\approx 880\mu s$; nous avons reussi à recouvrir $\approx 552\mu s$ de communication par du calcul, ce qui correspond au temps d'aller-retour ($760\mu s$) moins le surcoût estimé ($210\mu s$).
- Les durées minimales d'exécution de messages différentes de taille nulle sont assez proches. En effet le surcoût de traversée de la couche de communication est plus important que celui du traitement de données (pour le cas des messages considérés). L'écart entre les courbes à la saturation donne donc le temps de traitement de 256 octets.

On conclut donc qu'il est possible d'exploiter un recouvrement des communication par des calcul. Cependant ceci est fait par un effort additionnel de programmation où le programmeur est responsable de la bonne découpe calcul-communication et de leur synchronisation. L'emploi de la multiprogrammation légère réduit cette complexité de programmation en attribuant à un *thread* la tâche de réaliser les communications et à un autre les calculs. Le recouvrement est donc assuré par la multiprogrammation. Nous détaillerons cet aspect dans les chapitres suivants.

3.7 Bilan

La programmation parallèle par échange de messages s'est révélée propre au développement d'applications parallèles de grande taille sur des machines paral-

lèles comme des grappes de station. Un consensus s'est dégagé sur un ensemble d'opérations et un standard a été défini : MPI. Des implantations sont disponibles chez les industriels et dans le domaine public.

Des indicateurs de performances existent pour aider le programmeur à évaluer l'efficacité d'un programme. Aujourd'hui la limitation de ces bibliothèques provient de ce qu'elles ont été conçues pour la communication entre processus lourds ce qui rend difficile l'exploitation du parallélisme interne à un multiprocesseur mais aussi l'utilisation de ce parallélisme pour améliorer les communications. Cette « lourdeur » limite son emploi au domaine des applications régulières ou irrégulières à gros grain. C'est pourquoi de nombreux projets se sont penchés sur le problème de l'utilisation de *threads* et de la possibilité de les faire communiquer à distance. Le problème qui se pose alors est celui de l'intégration d'une bibliothèque de communication et d'un noyau de multiprogrammation légère.

4

L'intégration des *threads* et des communications

« Toute puissance est faible à moins que d'être unie »
Jean de La Fontaine, Le vieillard et ses enfants

L'intérêt d'associer *threads* et communications est d'obtenir un mécanisme simple réalisant automatiquement le recouvrement calcul-communication. Il faut se souvenir que cela a été la raison de l'invention de la multiprogrammation à la fin des années 60. Déjà à cette époque les délais de commande des périphériques et des transferts étaient grands devant les temps de cycle du processeur. Les informaticiens avaient déjà mis en évidence qu'un programme n'était qu'une suite de transferts et de calculs et que les transferts et les calculs pouvaient être fait en parallèle (par exemple écriture sur un disque et le calcul suivant la requête d'écriture). Quand cela n'était pas possible, on pouvait exécuter une phase de calcul d'un autre programme. Ces constatations était à la base de tout le travail théorique autour du concept de processus et de tout le travail technique conduisant aux systèmes multiprogrammés actuels. Marier *threads* et communications a donc pour objectif de fournir les mêmes facilités pour la programmation d'applications parallèles efficaces.

Ce chapitre débute par la présentation des divers mécanismes de mise en œuvre efficace de la communication en insistant plus particulièrement sur la réception de messages. La section suivante discute les problèmes d'intégration à un noyau de *threads* d'une bibliothèque de communication vue comme une « boîte noire », qui n'est pas toujours *thread aware* ou *thread safe*. Les dernières sections présentent ATHAPASCAN-0 et des expériences équivalentes. Elles sont suivies d'un bilan.

4.1 La réalisation des communications

Au même titre que les processus, les communications doivent être allégées, c'est à dire rendues plus efficaces. Cette opération est aujourd'hui critique car leur implantation actuelle ne permet pas d'exploiter des réseaux à haut débit tels Myrinet[24], EtherGigabit[5], etc. Le débit offert par le système au niveau applicatif est très inférieur à celui offert par le réseau. Cette dégradation provient des protocoles classiques (TCP/IP) et de leurs implantations initialement conçues pour des réseaux très hétérogènes, à faible débit, peu fiables et de grand diamètre. En particulier ces implantations sont inappropriées pour des réseaux à haut débit, fiables et de petit diamètre comme ceux des grappes SMP. Du point de vue du calcul parallèle distribué, cette inadéquation est accrue par le fait que le système considère généralement une communication comme une opération d'entrée/sortie « lente » impliquant la désactivation du processus et aucune urgence de traitement. La cure d'amaigrissement porte sur les aspects de l'implantation de la communication au niveau :

- Interface réseau-ordinateur : Ces interfaces ont pour but de décharger le processeur des transferts de données entre mémoire et réseau. De plus en plus, ils prennent à leur charge une partie des protocoles de communication SCI[99], Myrinet[24].
- Interface réseau-système application : assurer les communications au niveau du système devient critique du fait des surcoûts d'appel du système. La tendance actuelle est une approche où le système est responsable du partage du réseau entre applications et où les applications sont responsables de la réalisation de leurs communications en accèdant directement à la commande du réseau.

La réalisation des protocoles de communication au niveau utilisateur peut conduire à un gain d'efficacité considérable [158]. Plusieurs projets emploient cette solution pour exploiter de manière efficace les capacités offertes par les réseaux à haut débit [107] [132] [126], [165] [164] .

Cependant le partage du réseau entre applications est toujours un problème. En effet le noyau de communication doit non seulement s'occuper du multiplexage et du démultiplexage des communications issues des différents processus du réseau mais aussi de réaliser une protection d'accès, nécessaire pour éviter qu'une erreur compromette le système entier. Les projets comme Fast Messages [126], BIP [132] et U-NET [165] attaquent ce problème.

Dans ce qui suit nous allons décrire les interactions avec le réseau qui cadencent les communications et les différentes manières de mettre en œuvre ces interactions de communication. Le niveau de détail de cette présentation permet la mise en œuvre de ces méthodes tant au niveau pilote de périphérique qu'à tout autre niveau de la hiérarchie système[98].

4.1.1 Événement de communication

Seul l'interface réseau sait s'il est possible d'émettre ou nécessaire de recevoir. Ces états sont souvent fugaces, c'est à dire que la possibilité d'émission est liée à la détection de l'inoccupation du médium ou à des contraintes temporelles. De même, si quelque chose arrive sur le réseau, l'interface doit le capturer. D'une manière générale ces contraintes temporelles sont « affaiblies » par l'usage de tampons internes. Cependant le système ou l'application doivent acquérir les données à leur arrivée[1] et émettre au meilleur débit possible. Il est donc important d'asservir l'implantation de la communication à ces événements. Ceci suppose donc que le système (au sens large) puisse les détecter et réagir.

4.1.2 Interruption versus scrutation

Idéalement un système doit répondre aux événements externes dès qu'ils se produisent. On parle donc de sa **réactivité**, c'est à dire, du délai entre la manifestation de l'événement et sa prise en charge par le système. Pour cela, deux méthodes sont couramment utilisées : l'**interruption** et la **scrutation** (*polling*). L'emploi d'interruption est l'approche « classique » pour avertir de l'occurrence d'un événement asynchrone. Lorsque celui-ci se produit, l'interface réseau engendre une interruption au processeur et une procédure dite « traitant d'interruption » est exécutée. Dans le cas de la scrutation, il est nécessaire de vérifier périodiquement l'état de l'interface (procédure de scrutation).

La scrutation : La scrutation ne pose pas de contrainte forte sur le matériel et le système. Par contre, elle pose énormément de problème logiciel. Pour définir un mécanisme efficace de scrutation il faut un mécanisme peu coûteux qui permette de consulter l'interface avec une période d'échantillonnage appropriée au comportement temporel du réseau. Une fréquence trop élevée introduit des tests inutiles ; une fréquence trop basse ne permettra pas d'émettre au mieux ni d'acquérir tous les messages. La mise en œuvre efficace de la scrutation est délicate. Nous trouvons les solutions suivantes :

- le problème est renvoyé au niveau applicatif c'est à dire que l'application s'occupe de réaliser la scrutation à une fréquence appropriée pour assurer un niveau d'efficacité souhaité.
- un processus spécifique est dédié à cette tâche. L'efficacité de la scrutation va dépendre totalement du contrôle de l'ordonnancement des processus.
- la scrutation est faite via un traitement déclanché par l'interruption horloge. Ceci garantit une période fixe de scrutation. Ici on repose donc indirectement sur le mécanisme d'interruption.

[1] sous peine de perte de message.

Interruption : Le mécanisme d'interruption a été conçu pour une réaction efficace à un événement externe. Celui ci est observé en permanence par un circuit qui interrompt le processeur au moment approprié. C'est le mécanisme idéal pour obtenir une bonne réactivité.

Dans la pratique, ce n'est pas toujours vrai. Le mécanisme d'interruption a été conçu pour un fonctionnement interne au noyau du système. Aujourd'hui le mécanisme d'interruption offert au niveau applicatif est lourd car il suppose un prétraitement au niveau du noyau pour détecter le processus concerné, une simulation logicielle de l'interruption sur les processus suivie d'un délai d'ordonnancement variable pour le réactivation de celui-ci. Un mécanisme efficace au niveau système peut se révéler inefficace à un autre niveau.

L'efficacité d'une interaction avec le réseau ne dépend donc pas seulement de la réactivité du mécanisme utilisé mais de sa mise en œuvre et du nombre d'activations nécessaires pour effectuer un travail de communication donné avec une efficacité donnée. Par exemple, si l'interruption n'est plus engendrée à l'arrivée d'un message mais à chaque arrivée d'un octet, le système peut être « écroulé » par le nombre d'interruption à traiter.

Contrainte d'utilisation : Le cas défavorable pour l'utilisation d'interruptions correspond à des inter-temps entre événements voisin du coût du mécanisme[2]. En ce qui concerne la scrutation, il correspond à un taux d'échec trop important dû à une fréquence inappropriée. C'est qui se produisait sur notre machine IBM SP1 (interface TB2) avec la version prototype de MPI (MPI-F [79]) où il était possible de faire fonctionner MPI en mode interruption ou en mode scrutation. Dans le premier cas, le nombre d'interruptions engendrées pour une communication MPI dégradait la communication élémentaire de façon intolérable. Dans le second cas, la période de scrutation déclenchée par une interruption horloge étant trop importante. L'obtention d'une efficacité raisonnable était donc à la charge de l'application.

Pour diminuer la dégradation il est intéressant, si l'interface réseau le permet, de retarder le traitement de façon à traiter plusieurs messages à la fois pour le coût d'une seule exécution du mécanisme. C'est ainsi que l'on peut mettre un délai de garde, déclenchant la scrutation si celle ci n'est pas faite au bout d'un certain temps. On peut aussi mélanger les deux mécanismes. Dans le cas de séries d'événements très proches séparées par des durées importantes, il est possible de traiter le premier événement par interruption et les autres par scrutation [114]. C'est ce qu'il aurait été nécessaire de faire pour le SP1. De nombreux projets explorent ces compromis : EARTH-MANA [97], J-machine, [123], Remote Queues [27], et MIT Alewife [1].

[2] c'est généralement vrai pour des petits messages.

4.1.3 Le traitement de messages

L'acquisition des messages pose un problème spécifique. En effet, il est nécessaire de localiser l'emplacement mémoire où celui-ci doit être mis. Ce n'est pas le cas pour une émission qui fournit cette information. Dans la technologie « classique » des communications (TCP/IP, par exemple), le message est acquis dans un tampon intermédiaire du système ou rejeté faute de tampon. L'application vient retirer le message des tampons. Cette méthode simple est devenue inacceptable car :

- la recopie systématique du message interdit l'exploitation efficace des réseaux haut débit.
- la réactivité obtenue pour une application de calcul ou de mouvement de données distribuées est faible.

C'est pourquoi nous présenterons surtout les nouvelles approches élaborées dans le cadre de la multiprogrammation légère. Elles sont au nombre de trois : les **messages actifs**, le mécanisme d'***upcall*** et celui de ***pop-up***. Avant de les présenter, nous allons brièvement décrire la façon dont un message peut être acquis et les contraintes associées. Généralement tout message est véhiculé sur un réseau en un ou plusieurs paquets élémentaires dont le premier contient la description du message. Le premier paquet doit être reçu et décodé. Il doit être possible de déduire le nombre de paquets suivants et la zone mémoire où les mettre avant leur arrivée :

- si c'est possible, on reçoit les paquets dans leurs zone de destination finale. On parle de communication zéro-copie.
- si c'est impossible, il y aura copies des premiers paquets dans des tampons. Un mécanisme de contrôle de flux doit interdire de se trouver dans le cas où aucun tampon n'est disponible pour y mettre un paquet.

Une méthode simple est de n'émettre que le paquet descripteur. Le reste est émis à la réception d'un acquittement qui n'est envoyé que lorsque la zone de réception est connue. Elle a l'inconvénient de nécessiter plusieurs allers et retours dans le réseau pour l'émission d'un message. Toutes les méthodes suivantes d'acquisition des messages sont conçues pour s'adapter au mieux des possibilités du réseau sous jacent.

Les messages actifs

Le fondement de base des **messages actifs** (MA) [164] est que l'arrivée d'un message provoque l'exécution d'une procédure dont le message est le paramètre. Typiquement un message actif est composé de l'identification de l'émetteur, du nom de la fonction traitante du message actif (la procédure à exécuter à l'arrivée), et par un certain nombre de paramètres pour cette fonction (de zéro à quatre). Les messages actifs sont implantés par trois primitives : **request**, **get**, et **poll**.

Malgré la similitude avec l'appel de procédure à distance (RPC[3]), le message actif a un objectif différent : son rôle principal est d'extraire les messages de l'interface réseau et de les insérer le plus vite possible dans le calcul. La détection de l'arrivée du message actif peut être faite soit par interruption soit par scrutation, selon le schéma de la figure 4.1.

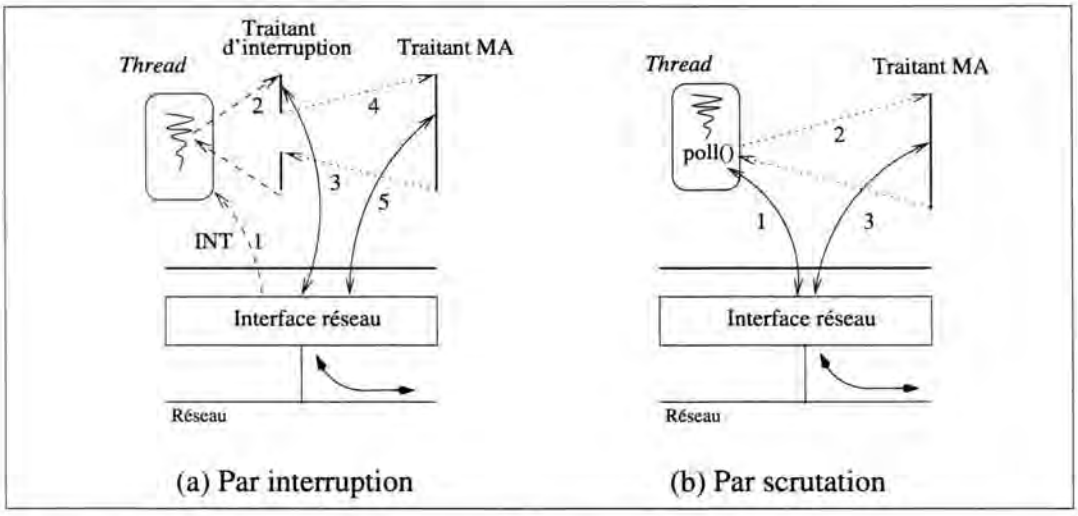

Figure 4.1 *Fonctionnement schématique des messages actifs*

En utilisant la méthode « par interruptions », l'interface réseau interrompt le *thread* en exécution lorsqu'un message arrive (pas 1 sur la figure 4.1.a). Ceci déclanche un traitant d'interruption (pas 2) lequel lit le paquet descripteur du message de l'interface réseau. Il en extrait les informations nécessaires, dont la procédure à exécuter (pas 3), et appelle cette procédure (pas 4). Celle-ci transfère les données du réseau vers une zone mémoire conventionnelle par l'opérateur **get** (pas 5).

Le bon fonctionnement suppose que le premier paquet étant acquis, la durée de la prise en compte de l'interruption jusqu'à l'exécution du premier **get** permette d'acquérir les autres paquets à la volée. Dans le cas contraire, les paquets doivent être mémorisés. Si cela n'est pas fait, le **get** engendre un acquittement qui provoque l'émission des paquets qu'il attend.

Le cas « par scrutation » est similaire. Ici, l'application est responsable de la scrutation du réseau de façon périodique par un appel explicite à la procédure de scrutation (***poll()***). L'interface réseau est alors consultée (pas 1 sur la 4.1.b), et en présence d'un message, le traitant de message actif est exécuté (pas 2). Comme dans le cas précédent pour les primitives de type **get**, un transfert des données est fait entre l'interface réseau et une zone mémoire (pas 3). Dans le cas de la scrutation, une copie dans des tampons est donc toujours nécessaire. Elle peut ne concerner que le paquet descripteur si un acquéreur (get) déclenche l'envoi des autres paquets.

Dans les deux méthodes, interruption ou scrutation, le traitant de message actif s'exécute avec la pile d'exécution du *thread* actif à cet instant. C'est le grand

[3]Remote Procedure Call.

avantage apporté de cette méthode : il élimine tous les coûts associés à l'ordonnancement d'un *thread* spécifique. Cependant, si le traitant du message réalise des attentes ou se bloque, il peut provoquer des inter blocages. Pour les éviter aucun appel potentiellement bloquant ne peut être fait, aucun mécanisme de synchronisation (verrous, variables de condition) ne peut être utilisé par un traitant. C'est le défaut de cette solution. S'il est impératif qu'un traitant réalise une opération bloquante il doit déléguer ces actions critiques à des *threads* « normaux » via un mécanisme approprié comme les **continuations** [55].

En dernier ressort le débit d'acquisition de message est complètement défini par la somme de la durée de déclenchement du traitant et de la durée de traitement. La méthode d'acquisition par message actif est donc très liée à l'implantation d'un mécanisme d'interruption efficace. Ce mécanisme est généralement fournit au niveau natif d'un processeur ou dans certains noyaux temps-réel. La plupart du temps, (comme dit en introduction du chapitre) la prise en compte d'une interruption au niveau applicatif est très lourde, ce qui limite fortement l'efficacité du message actif compte tenu des contraintes de programmation qu'il induit au niveau du programmeur ou des contraintes de mémorisation au niveau réseau. C'est pourquoi d'autres méthodes ont été proposées.

Le mécanisme de messages actifs a été initialement employé lors de l'implantation de SplitC [49] et dans la conception de systèmes comme TAM (Thread Abstract Machine)[50], J-Machine[123]. Le principe des messages actifs a été encore utilisé par Fast Messages [126] et par MADELEINE [26].

Le mécanisme d'*Upcall* (appel ascendant)

Dans cette méthode un *thread* spécialisé est responsable du traitement de tous les messages qui arrivent au nœud. Ce *thread* est souvent nommé **démon de communication**. Le démon est toujours en attente de message. À l'arrivée du paquet initial d'un message, il est activé. Il sélectionne la procédure à exécuter de la même façon que pour un message actif et l'appelle. Celle-ci acquiert le reste du message. À la fin le démon se met en attente du prochain message. Par rapport au message actif, la méthode d'appel ascendant (*upcall*) diminue les contraintes de programmation pesant sur la procédure de traitement de message.

Le traitant est exécuté maintenant dans un contexte à part, celui du démon, ce qui rend possible l'utilisation de mécanismes classiques de synchronisation entre *threads* nécessaires pour assurer la modification atomique de données ou la signalisation d'un traitement à faire de façon différée. Pendant que le démon est bloqué, aucun autre message ne peut être reçu. Il est donc nécessaire de contrôler finement la durée de transit en section critique. De plus l'usage d'opérateur bloquant (P sur sémaphore, attente de condition) est exclu du fait des risques d'interblocage. Il est aussi généralement impossible d'effectuer des communications qui risqueraient d'être bloquantes. On retrouve donc, sous une forme allégée, des contraintes voisines de celles associées aux message actifs.

Le bon fonctionnement de cette méthode est aussi assujetti à des contraintes de réactivité. Une prise en compte au plus tôt de l'arrivée d'un message exige d'être capable de réquisitionner un processeur pour l'exécution du démon (figure 4.2.a).

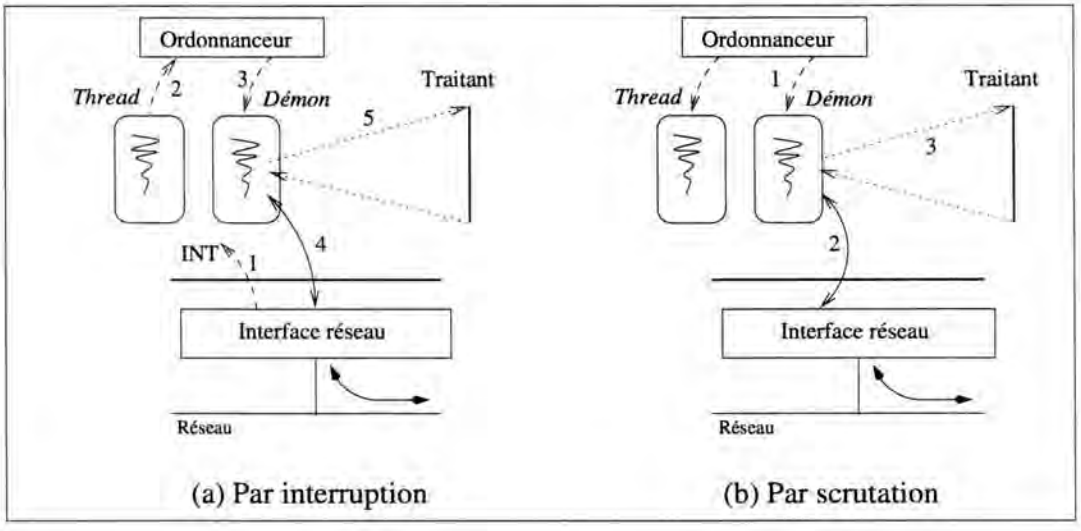

Figure 4.2 *Fonctionnement schématique du mécanisme d'*Upcall

Lorsqu'un message arrive, une interruption est générée (pas 1), ce qui provoque l'appel de l'ordonnanceur. Le *thread* en exécution est arrêté (pas 2) au profit du démon (pas 3). Le paquet initial est extrait de l'interface réseau (pas 4) et le reste peut être acquis par le traitant (pas 5). Tout retard dans le démarrage du démon imposera une mémorisation partielle du message. Si le retard ne peut être borné du fait d'une multiprogrammation n'activant le démon qu'après tous les *threads* prêts à l'arrivée du message, il faut, comme dans la méthode précédente, introduire un contrôle du flux des paquets.

Si le démon ne peut être déclenché par interruption, il devra tester périodiquement la disponibilité d'un message (fig.4.2.b). Dès que le démon devient actif (pas 1), il consulte l'interface réseau pour vérifier la présence de messages (pas 2). S'il y en a, le traitant est exécuté (pas 3). Son exécution dépend complètement de l'ordonnancement des *threads*, c'est à dire de la possibilité d'indiquer que le démon devra tourner sur une certaine base temporelle. Ceci n'est généralement possible que sur un système offrant une gestion d'échéances et un mécanisme de priorité.

Le débit d'acquisition de messages dépend du délai d'ordonnancement augmenté de la durée de traitement d'un message. Ainsi les contraintes pesant sur un mécanisme de traitement d'interruptions, réel ou simulé, se sont déplacés sur l'ordonnanceur du système. Un exemple de système basé sur le concept d'*upcall* est le noyau exécutif Panda [20].

Le mécanisme de *PopUp*

La méthode suivante est destinée à éliminer les contraintes de programmation pesant sur la procédure d'acquisition/traitement du message et à rendre le débit d'acquisition du message indépendant de la durée de ces procédures.

Un *thread* est créé pour acquérir et traiter tout nouveau message. Comme tout nouveau message est traité par un *thread* dédié indépendant des autres, celui-ci peut se synchroniser à leur gré sans influer sur l'acquisition des nouveaux messages. Le débit d'acquisition des messages est donc uniquement dépendant du délai de déclenchement de la création d'un *thread* à l'arrivée d'un message augmenté du coût de création du *thread* et du coût de recopie du message chez le *thread* créé.

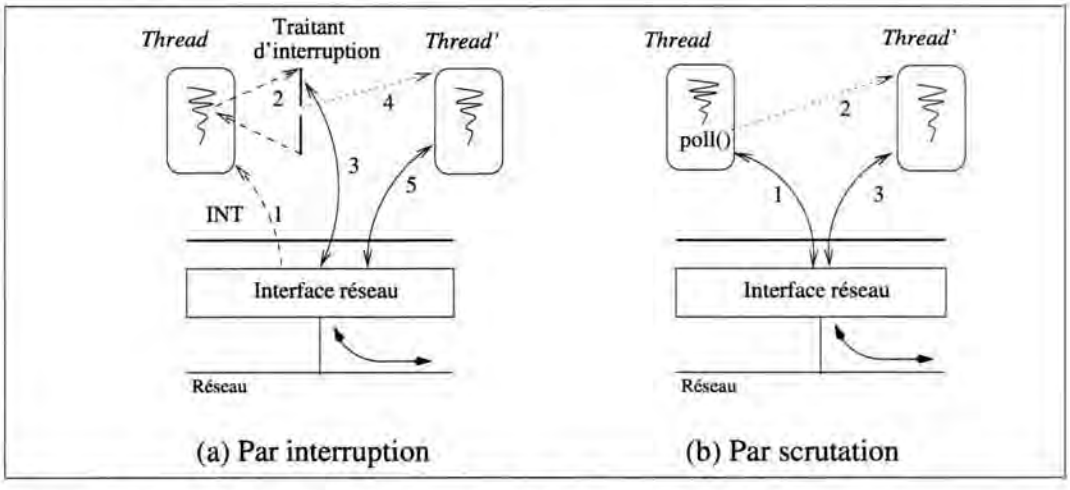

Figure 4.3 *Fonctionnement schématique du mécanisme de* PopUp

L'implantation de cette méthode est soumise à la même contrainte que les autres. Il faut que le délai entre l'arrivée du message et l'acquisition pour le traitant soit aussi court que possible pour éviter le recours à des tampons intermédiaires ou au contrôle de flux du message. La création du *thread* peut être assurée par traitant d'interruption (pas 4, fig. 4.3.a) ou par l'exécution de la procédure de scrutation (pas 2, fig. 4.3.b). Ce qui est important c'est que le délai d'extraction du message soit le plus court. Une façon d'assurer cette rapidité est de faire l'extraction avant la création du *thread* dédié au message. Sinon il faut garantir que ce *thread* sera « prioritaire » jusqu'à ce qu'il ait acquis le message (get). Cette méthode rend le débit de messages acceptés indépendant du traitement des messages. Ceci peut avoir les conséquences suivantes :

- Le nombre de *threads* à un instant donné peut devenir très grand dès que le débit de traitement des messages est inférieur au débit d'arrivée.
- Un nombre trop important de *threads* peut saturer les ressources du système (mémoire insuffisante) ou l'écrouler si le système se dégrade à la charge (surcoût d'ordonnancement, écroulement de la pagination, etc).

- l'ordre FIFO de traitement des messages est difficile à garantir. Par exemple, un processus A envoie deux messages m_1 et m_2, dans cette ordre, vers un processus destinataire B. À la réception, deux *threads* seront créés. Ils s'exécutent en parallèle ou de façon entrelacée. L'exécution du traitement de m_2 après celui de m_1 doit être explicitement programmée si nécessaire.

Le méthode de *popup* est plus flexible et plus facile à utiliser. C'est un grand avantage pour le programmeur. Elle pose cependant des contraintes importantes sur l'implantation des *threads* (création) et l'ordonnancement. En particulieur, celui-ci doit offrir un contrôle de priorités pour que cette méthode soit efficace. En dépit des lacunes des noyaux de *threads* actuels, plusieurs environnements de programmation, comme Nexus [76], x-kernel[98], horus [163], ATHAPASCAN-0b [86] utilisent ce concept pour effectuer le traitement des messages. La section suivante présente l'état de l'art de l'intégration de la communication à la multiprogrammation légère.

4.2 État de l'art

La plupart des implantations actuelles ont choisi d'intégrer des noyaux de multiprogrammation légère existants à des bibliothèques de communications existantes (MPI[88], PVM[85]). Certains projets ont choisi de reconcevoir un des deux éléments. Un seul projet, Panda [20], a reconçu les deux parties.

Les projets qui ont choisi de réutiliser une bibliothèque de communication ont dû faire face aux problèmes posés par l'utilisation d'une « boîte noire » n'ayant pas été conçue pour être utilisée concurrement par des *threads* et offrant rarement une implantation efficace. Par ailleurs, les projets diffèrent par le niveau auquel l'intégration de la multiprogrammation légère et de la communication légère a été fait. Dans les cas les plus simples, on a affaire à une simple intégration de la communication à la mécanique de multiprogrammation. À l'opposé se trouve l'approche essayant de proposer une interface applicative rendant la distribuition transparente à l'utilisateur. Quel que soit le niveau choisi, l'architecture d'intégration est voisine et fait face à des problèmes et solutions communes.

4.2.1 Architecture et niveau d'intégration

L'architecture d'intégration est identique pour tous les projets (fig. 4.4). Une couche logicielle assure l'interface entre l'application placée sur un processus et les noyaux de communication et de multiprogrammation légère. Cette couche assure en outre la coopération entre ces deux noyaux. Elle permet aux *threads* placés sur un processeur virtuel d'un nœud de communiquer avec des *threads* d'autres nœuds par l'intermédiaire d'un sous ensemble des fonctions de la bibliothèque de communciation. La multiprogrammation légère est utilisée pour le recouvrement des communications par le calcul. Ainsi, par exemple, Nexus[76], Panda [20], ATHAPASCAN-0a

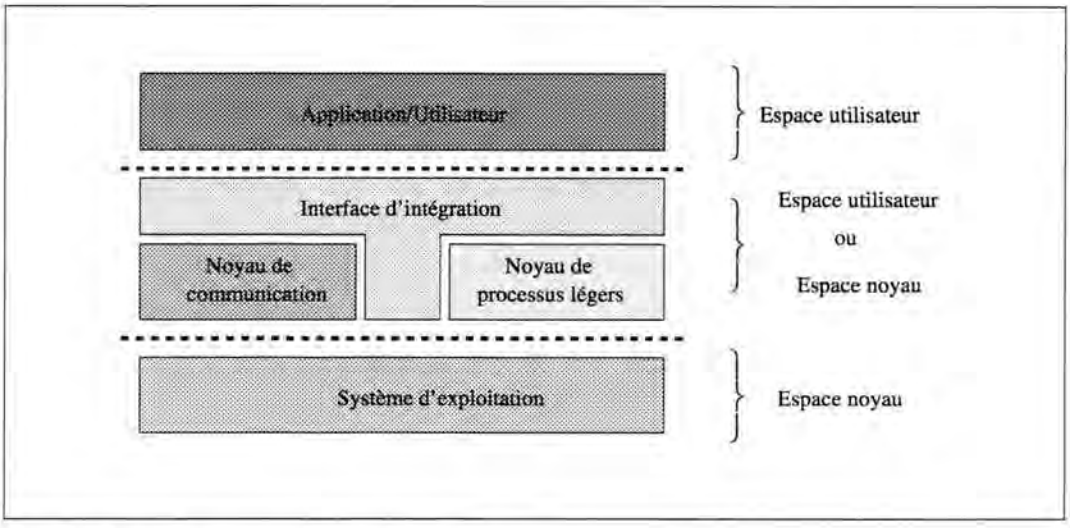

Figure 4.4 *Architecture « classique » d'un noyau exécutif*

[42], PM^2 [120] offrent un mécanisme de communication de type *popup*. ATHAPASCAN-0b [86] offre en plus des canaux bi-points entre *threads*. D'autres projets ont cherché à proposer une interface masquant plus ou moins les problèmes liés à la distribution, comme :

- le placement dynamique de *threads*.
- la migration de *threads*.
- le partage et la synchronisation d'accès à des données distantes.

Dans une telle approche, la couche d'intégration doit offrir une nomenclature globale des *threads* et des objets associés. C'est le cas des projets Chant [90] ou Compositional C++ [37]. Aller plus loin exige des abstractions de plus haut niveau que l'union d'une interface POSIX *threads* et d'une interface MPI (ATHAPASCAN-1 [82], ORCA [12]). Les fonctionnalités de partage des objets, et de régulation de charge exigent des coopérations entre les différentes couches des nœuds. Notons que certains projets, comme RThreads[56][57][58], proposent d'atteindre cet objectif par un noyau de gestion de mémoire virtuelle distribuée. Dans ce qui suit, nous nous restreindrons au problème de fournir aux *threads* les deux services de base communs à tous les projets quelque soit le niveau d'intégration choisi. Le premier est celui de l'accès concurrent des *threads* à la communication. Le deuxième porte sur l'interaction communication-multiprogrammation légère nécessaire à la bonne progression des communications. C'est le problème de « tester et faire avancer » les communications.

4.2.2 Accès concurrent

L'implantation des noyaux de communication nécessite généralement le maintien des diverses informations liées à son état interne de fonctionnement. Ces in-

formations sont encapsulées dans des objets opaques[4]. La mise à jour, ou même la lecture, de ces informations doit être atomique. Dans une bibliothèque utilisée par un seul *thread* l'atomicité est garantie de fait. Dès que plusieurs *threads* peuvent s'entrelacer au cours de l'exécution de la bibliothèque, il y a risque de corruption et d'erreur (fig. 4.5.a). La plupart des bibliothèques MPI disponibles (LAM, MPICH, CHIMP, etc), malgré la définition d'une interface *thread-safe* par le standard MPI, présentent ce défaut.

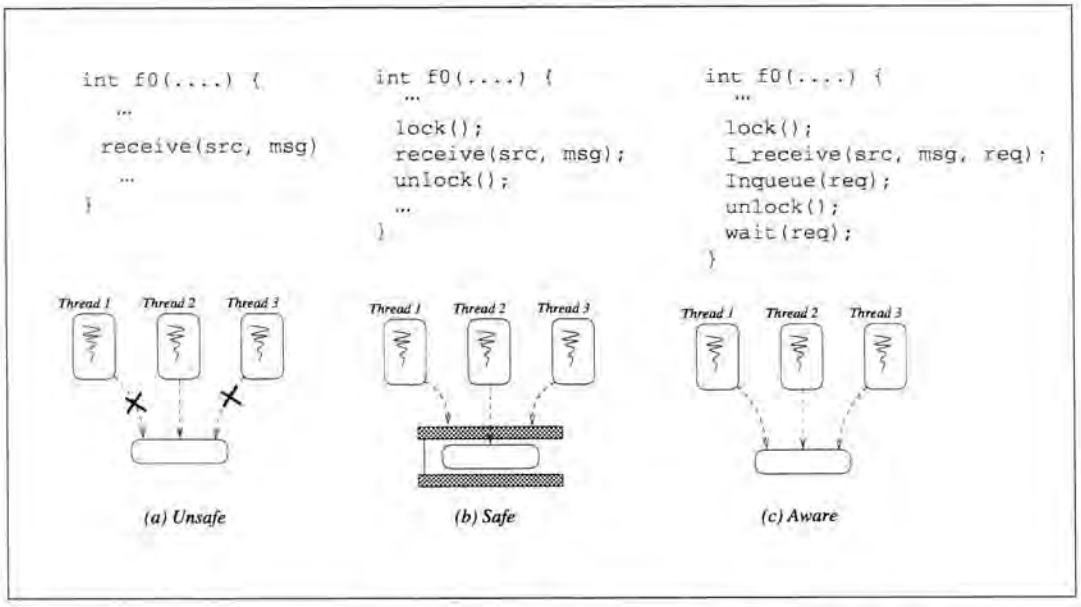

Figure 4.5 *Aspects* thread unsafety, thread safety *et* thread awareness *de MPI*

On rend le noyau de communication *thread-safe* en garantissant son exécution en exclusion mutuelle (fig. 4.5.b). Ce verrouillage peut être de longue durée et nuire à la vivacité du système. Il peut engendrer des interblocages. En particulier si les *threads* sont implantés en espace utilisateur (modèle N :1), le blocage d'un *thread* à l'intérieur du noyau de communication provoque le blocage de tous les autres *threads* exécutés par le processus lourd « portant » le *thread* bloqué. Il faut pouvoir rendre les opérateurs de communication *thread aware*. Deux approches voisines sont possibles. Elles supposent que l'on puisse soit tester la faisabilité d'une communication soit disposer d'opérateurs non bloquants.

Dans le premier cas, on délègue à une procédure de scrutation la tâche d'effectuer les communication quand elles sont possibles. Le *thread* demandeur transmet, via une file, une description de la communication à faire. Une procédure de scrutation périodiquement appelée exécute celles qui ne sont pas bloquantes à ce moment. C'est le cas, par exemple, des appels système *fcntl* et *ioctl* d'UNIX.

L'autre solution repose sur l'existence d'opérateurs de communication non bloquante. Une communication bloquante est remplacée par une non bloquante. À ces

[4]Un objet opaque est un objet dont la structure interne nous est inconnue. L'accès aux membres est fait au travers de fonctions spécifiques.

opérations non bloquantes sont associées des objets appelés **requêtes** (*request*), qui permettent de savoir si une requête est terminée. Une procédure de scrutation périodiquement appelée examine les requêtes terminées.

Lorsqu'une requête est terminée, sa fin est signalée au *thread* l'ayant initiée, lequel est donc débloqué et peut suivre son exécution. Ce fonctionnement est résumé par les procédures de la figure 4.6. On a bien réussi à réduire le blocage lié à l'exclusion mutuelle des communication en mémorisant dans une file les communications bloquantes.

Dans ce qui suit on détaillera le cas des opérateurs non bloquants que nous avons rencontrés en MPI.

```
\* interface de communication *\

int NewReceive(src, msg) {
  ...
  lock();
  I_receive(src, msg, req);
  Inqueue(req);
  unlock();
  wait(req);
}
```

```
\* Procédure de scrutation
 * (Traitant d'interruption)
 *\

void Poll(void) {
  ...
  lock();
    forall req in Queue
      if (probe(req))
        Dequeue(req);
        signal(req);
  unlock();
}
```

Figure 4.6 *Fonctionnement général de la scrutation*

4.2.3 La problématique « tester et faire avancer »

Le problème est de garantir l'avancement d'une requête de communication lancée lors de l'appel non bloquant. En d'autres termes, il faut inspecter régulièrement la file de « communications en cours d'exécution » (requête). Idéalement il faut réagir à une requête de communication dès que le réseau est prêt à émettre ou dès qu'un message arrive. En règle générale, les noyaux de communications tels que MPI et PVM ne font pas remonter un signal d'interruption vers l'application. En effet, la fin d'envoi, ou la vérification de l'arrivée d'un message, n'est faite que lors d'un appel à une primitive prévue pour cela. Il est alors nécessaire de venir périodiquement **« tester et faire avancer »** les communications.

La fréquence de « tester et faire avancer » dépend du régime de communication de l'application. Si l'on scrute le réseau à une fréquence trop élevée, on volera du temps d'exécution aux *threads* de calcul. On provoquera des commutations de contexte d'autant plus inutiles que la fréquence d'envoi de messages est faible. Par contre, si l'on scrute à une fréquence trop basse, les délais de communication seront

allongés en conséquence et ils provoqueront éventuellement des attentes de calculs. Cette question de la scrutation (« tester et faire avancer ») est la même que celle discutée à la section 4.1.2. La seule différence est que ce problème a été déplacé du niveau interface réseau/traitant de message vers le niveau noyau de communication/application utilisateur.

Au niveau application utilisateur, la mise en œuvre de la scrutation dépend des facilités fournies par le système d'exploitation, par la librairie de *threads* et par le noyau de communication. En effet, l'appel de scrutation peut être fait par :

- l'application elle même.
- un *thread* spécialisé (démon de communication).

Dans le premier cas, l'application appelle la procédure de scrutation. L'avantage de cette solution est de libérer la couche d'intégration de ce problème. L'inconvénient est que le programmeur doit choisir les bons endroits et moments pour exécuter la scrutation. L'insertion automatique d'appels à la procédure de scrutation par un compilateur, ou pré-processeur, est une solution encore hors d'atteinte de la technologie. Le programmeur hérite donc d'une tâche très délicate.

La deuxième possibilité est d'employer un *thread* spécialisé. La période de scrutation dépend des priorités et des politiques d'ordonnancement mises à disposition par le système d'exploitation et/ou par la librairie de *threads*. Ainsi dans le cas d'une politique *sans temps partagé - sans priorités*, le *thread* qui exécute la scrutation a la même priorité qu'un *thread* utilisateur. La période de scrutation n'est pas prévisible (durée d'exécution cumulée des *threads* prêts). Dans le cas d'une politique *temps partagé sans priorité*, le temps partagé fixe une borne supérieure au délai d'attente d'un *thread* : il est donné par le produit charge (nombre de *threads* prêts) par le *quantum*. On peut donc avoir une borne supérieure à la période de scrutation. Dans le cas d'une politique *Temps partagé avec priorités*, un *thread* prioritaire a un délai d'attente borné par le *quantum*. La période de scrutation dépendra donc de la priorité du *thread* de communication par rapport à celle des utilisateurs.

A l'heure actuelle, la plupart des systèmes UNIX (Solaris, Linux, AIX, IRIX, etc.) implantent les *threads* en suivant un modèle 1 :1, c'est à dire, *threads* noyau (*kernel thread*). Ceci s'explique par le fait que ce modèle est simple et offre la capacité d'exploiter le parallélisme réel présent sur des machines multiprocesseurs. L'ordonnanceur UNIX « classique » essaie de garantir que les processus (et les *threads*, par conséquent) disposent des processeurs pendant des durées égales lors d'une période donnée. La valeur du *quantum* et l'affectation des priorités sont sous le contrôle du système. On peut estimer donc qu'un *thread* parmi k *threads* utilise un pourcentage $1/k$ du temps processeur. La priorité d'un *thread* croît avec son temps d'inactivité. Ainsi, un *thread* effectuant des entrées-sorties devient plus prioritaire afin de rattraper son « retard » de calcul.

Dans un tel cas, le *thread* spécialisé (démon) est considéré comme un *thread* sans prérogatives particulières : il a droit à une part équitable ($1/k$) de temps

d'exécution. Sa priorité instantanée dépendra de la durée des blocages sur les entrées/sorties et du temps de calcul passé à scruter. Ceci a un effet direct sur la **réactivité** du système. La période de scrutation ne peut pas être inférieure au *quantum* en présence de *threads* de calcul. Par contre, le démon peut monopoliser inutilement le processeur (du fait du « retard » accumulé) si la fréquence de communication est basse.

Un autre point concerne l'efficacité de la scrutation. Dans le pire de cas il s'agit d'explorer la liste des requêtes possibles pour trouver celles faisables ou faites. Le coût croit avec la taille de la liste. On mesure ici l'intérêt d'un dispositif actif (message actif, *upcall*) qui permettrait d'exécuter une procédure bien déterminée lorsque les communications sont terminées ou deviennent faisables.

4.3 Les noyaux exécutifs existants

Les exemples d'intégration de la communication à un noyau de multiprogrammation légère ont fait face au même problème d'utilisation de bibliothèques ne remontant pas au niveau applicatif fonctionnalités de type message actif, *upcall*, ou *popup*. Bien qu'existant de façon interne au niveau système ou bibliothèque, ces mécanismes ne sont pas disponibles. L'interface est donc une interface classique de type send/recv (cf. chapitre 3). Un point important de tous ces projets a porté sur la reconstruction de ces fonctionnalités au niveau des *threads* par la mise en place d'une scrutation appropriée.

4.3.1 Nexus

Le noyau exécutif **Nexus** [76], développé au sein de l'Argonne National Laboratory, Illinois (USA), a été conçu dans le but de fournir un support à l'exécution d'applications parallèles sur architectures distribuées hétérogènes. **Nexus** offre une série de fonctionnalités plutôt destinées à des compilateurs de langages parallèles qu'à des programmeurs d'applications parallèles. Essentiellement, ces fonctionnalités concernent la multiprogrammation légère et la communication.

Nexus présente plusieurs niveaux d'abstraction, ce qui permet la réalisation d'une application **Nexus** en utilisant plusieurs langages de programmation, plusieurs exécutables, différents types de processeurs et différents protocoles réseau. **Nexus** a été utilisé pour la réalisation de compilateurs de langages parallèles, comme CC++ [36] et Fortran M [75], et comme bibliothèque de communication pour MPICH [88], nPerl [74] et CAVEcomm [48]. Le project *Globus* [73], qui définit un environnement pour la programmation distribuée de haute performance dans un cadre de *metacomputing* [146] , a pris Nexus base.

Nexus est organisé sur cinq abstractions fondamentales : les *nœuds*, les *contextes*, les *threads*, les *demandes de service distants* (*remote service request (RSR)*),

et les *pointeurs globaux* (*global pointers (GP)*). Une application Nexus s'exécute sur une machine virtuelle composée d'un certain nombre de *nœuds*, définis au moment du lancement de l'application. Cet ensemble peut évoluer dynamiquement soit par ajout soit par suppression. Les *contextes* sont créés à l'intérieur de chaque *nœud* et représentent un espace d'adressage similaire à celui d'un processus UNIX, où les *threads* sont créés et exécutés. Plusieurs *contextes* peuvent être créés sur un même *nœud*, et chaque *contexte* peut avoir plusieurs *threads* en son sein.

À l'intérieur d'un même contexte la communication est faite par mémoire partagée. L'identification du contexte destinataire sur un nœud distant est faite par l'abstraction de pointeur global. Le pointeur global est une structure de données qui contient une référence directe à un objet dans un contexte ou une référence indirecte. Dans ce cas, cette référence permet de localiser (au sens d'une adresse réseau) la prochaine référence. La seule primitive de communication offerte par *Nexus* est la *demande de services distants* (*remote service request, ou RSR*). Ce concept combine les concepts de message actif, *upcall*, ou *popup*. Si Nexus est utilisé sans *thread*, le fonctionnement est celui des messages actifs. En présence de plusieurs *threads*, un démon interprète un message comme une requête d'exécution de type *popup* (création de *thread* dédié), ou de type *upcall* (exécution par le démon).

Le point intéressant est la façon dont le gestionnaire de RSR sur un nœud détecte l'arrivée d'un message RSR. **Nexus** propose plusieurs implantations de RSR selon la possibilité et l'intérêt de modifier le système d'exploitation, et les fonctionnalités présentées par le noyau de processus légers et par le noyau de communication du système cible. Fondamentalement quatre solutions sont possibles :

Réception bloquante : cette méthode est possible lorsque le noyau de communication et le noyau de *threads* permettent à un *thread* de se bloquer sur une primitive de communication sans bloquer les autres *threads*. Cette solution considère donc un noyau de communication de type *thread aware* et l'emploi d'un noyau de multiprogrammation légère du type 1 :1 ou M :N. Dans ce cas, il est possible d'affecter dans chaque contexte un *thread* particulier pour gérer le traitement des requêtes RSR provenant d'autres contextes. Un tel *thread* reste bloqué jusqu'à ce qu'une demande de RSR arrive.

Scrutation à la demande : la scrutation du réseau est faite par appel explicite à une procédure, qui vérifie l'arrivée des RSR. Ainsi, il est possible d'insérer, à certains points de l'exécution des *threads* de calcul, un appel à cette primitive. La scrutation est donc à la charge de l'application.

Scrutation par un *thread* spécialisé : cette méthode est employée quand le noyau de communication ne permet pas aux *threads* de se bloquer sur une opération de communication sans provoquer le blocage du processus entier. Cela est notamment le cas de tous les noyaux de *threads* au niveau utilisateur. Un *thead* spécialisé vérifie l'arrivée des demandes RSR via des primitives non bloquantes. La réactivité du système dépend fortement des caractéristiques propres à l'ordonnanceur du système (politique d'ordonnancement, priorités,

préemption, etc...) qui détermine les moments où le *thread* spécialisé sera exécuté.

Basé sur interruption : cette technique repose sur la disponibilité d'une interruption lors de l'arrivée d'une requête RSR. On est dans un pur schéma de message actif avec les problèmes de programmation associée (cf. section 4.1.3). Le point critique est le coût de remontée d'une telle interruption du système jusqu'au processus cible (traitant de signal). Ce coût devient de plus en plus important d'autant que les latences réseaux diminuent.

Toutes ces solutions présentent des avantages et des inconvénients plus ou moins importants en fonction des caractéristiques propres aux systèmes cibles. Une très intéressante discussion et évaluation de ces politiques est présentée en [76]. Les différentes mesures de performances réalisées sur des grappes de monoprocesseurs par les développeurs de **Nexus** montrent que le surcoût introduit par **Nexus** pour l'envoi de messages de petite taille est relativement élevé (80% sur une machine SP1 par rapport à la bibliothèque de communication de base MPL). En fonction de cette caractéristique, **Nexus** s'avère plus adapté aux applications de calcul scientifique où de grands volumes de données sont manipulés. Cette même étude conclut que, en moyenne, la politique qui donne le meilleur résultat est celle de la scrutation par un *thread* spécialisé. Enfin, il y a un dérnier point favorable à cette solution : sa simplicité d'implantation et la portabilité offerte.

4.3.2 Chant

Le noyau exécutif **Chant** [89] a été développé avec l'objectif d'offrir un environnement de programmation parallèle pour des architectures à mémoire distribuée par l'extension des propriétés de base des *threads*. Le modèle de programmation de **Chant** est celui de la multiprogrammation légère où certaines primitives POSIX ont été étendues pour permettre leur utilisation indépendamment du *thread* qu'il soit local ou distant, comme la primitive *join* par exemple[5]. La principale contribution de *Chant* reste la capacité de communication directe entre *threads* par le biais de primitives de type *send/recv*. Les *threads chant*[6] sont aussi appelés **« talking threads »**. Parmi d'autres caractéristiques de **Chant** nous pouvons citer les opérations collectives à un groupe de *threads* (*ropes*) [91] et un mécanisme d'exécution de services à distance (RSR). Le noyau exécutif *Chant* est utilisé comme support d'exécution pour un langage data-parallèle dans le cadre du projet Opus [117].

Une application **Chant** est donc vue comme un ensemble de processeurs virtuels (processus UNIX) où s'exécutent des *threads*. Les *threads* que l'on souhaite faire

[5] les opérateurs en rapport avec la mémoire partagée (verrous, conditions) n'ont pas été étendus.

[6] Pour rendre possible la communication entre *threads* distants de façon transparente, **Chant** définit un nouveau type de *thread* connu par tout le système avec un nom global. Ces *threads* sont appelés *chanter*. Seul les *chanters* ont la capacité de communiquer entre eux en utilisant les primitives de communication de type *send* et *receive*.

interagir avec les *threads* d'autres contextes (processus), doivent être impérativement de type *chanter*. Ici, à nouveau, nous sommes confronté au problème de la détermination de l'arrivée d'une requête de l'extérieur. Le noyau exécutif **Chant** emploie une **scrutation au niveau utilisateur** et trois politiques ont été utilisées : *thread polls*, *server polls*, et *scheduler polls*.

La première solution, ***thread polls***, considère que la scrutation est faite par le *thread* qui réalise l'appel. Toutes les opérations de type synchrone sont donc traduites par un appel asynchrone suivi d'un test de complétude. Dans ce cas-là, le *thread* qui réalise l'opération est toujours dans un état prêt à exécuter. Lorsqu'il est activé par le système, il effectue le test de complétude. Si l'opération a été réalisée, le *thread* suit son exécution normalement ; dans le cas contraire, il libère le processeur pour un autre *thread*. Ceci nécessite de disposer d'un opérateur pour « passer la main » immédiatement (*yield()*) qui n'est pas toujours disponible[7] ou présente une sémantique imprécise. Cette solution a comme avantage d'être extrêmement portable et simple à programmer. La portabilité est assurée par l'existence des primitives d'envoi asynchrone et test, disponibles dans la plupart des bibliothèques de communication. L'inconvénient de cette solution est la génération de changements de contexte qui monopolisent le processeur au détriment de *threads* de calcul.

La deuxième solution, ***server polls***, consiste à créer un *thread* spécialisé qui sera responsable de la vérification de la complétude des opérations de communication effectuées par tous les *threads* utilisateurs (fig. 4.6). Une primitive de communication synchrone réalisée par un *thread* au niveau application est donc traduite en un appel à la version asynchrone de cette primitive, suivie de l'enregistrement de cette opération dans une structure de données accessible par le *thread* spécialisé, et finalement par un blocage en attente de la réalisation de cette opération. Le *thread* spécialisé, lorsqu'il est activé, parcourt cette structure en vérifiant l'avancement des opérations enregistrées. Quand une opération est finie, il débloque le *thread* associé. Cette solution évite des changements de contexte inutiles mais introduit le surcoût de vérification pour chaque exécution, de toutes les requêtes enregistrées. Un autre point délicat est que le *thread* spécialisé doit être exécuté à une fréquence appropriée.

Finalement, dans la troisième solution, ***scheduler polls***, l'ordonnanceur des *threads* a la tâche d'effectuer la scrutation. Celui-ci peut donc la faire à chaque commutation de *thread*. Le contrôle de la scrutation est bien meilleur que dans la méthode précédente et présente moins de test inutiles que dans la méthode où la scrutation est faite par les *threads* communicants eux-même. Un *thread* utilisateur en réalisant un appel de communication synchrone provoque l'exécution de la primitive asynchrone équivalente, l'enregistrement de cette requête sur une structure de données appartenant à l'ordonnanceur, et ensuite se bloque en attendant l'exécution de la primitive. L'ordonnanceur lors de son activation parcourt cette structure en cherchant un *thread* pour l'exécution. S'il trouve un *thread* bloqué en attente de communication pour lequel la communication est prête, le *thread* est réactivé ; sinon l'ordonnanceur

[7]Pas dans le standard POSIX.

continue sa recherche d'un *thread* prêt. L'avantage de cette solution est la réduction du nombre de changements de contextes et de tests de requêtes. L'inconvénient est la modification nécessaire de l'ordonnanceur de *threads*.

Les développeurs de **Chant** ont vérifié le comportement de ces trois politiques de scrutation par l'exécution d'un programme synthétique où plusieurs paires de *threads* distants réalisaient des calcul suivis d'une séquence de *send* et *recv* entre eux. Ils ont rapporté dans [89][90] un surcoût de 10% pour la solution *thread polls* par rapport à la solution *scheduler polls*. La solution *server polls* a donné les moins bons résultats. De ce fait, la solution retenue a été celle de *thread polls* car le surcoût introduit est compensé par la facilité de programmation. Ce résultat est contradictoire avec les résultats de **Nexus** [76]. Dans ces deux projets, les expériences ont été conduites sur des monoprocesseurs.

4.3.3 Panda

Panda [20] est un noyau exécutif portable conçu pour offrir un support général et flexible pour la réalisation d'environnements de programmation parallèle distribuée. Initialement prévu pour être le support exécutif du langage de programmation ORCA [12], sa souplesse a permis aussi son emploi pour l'implantation de PVM et du langage de programmation SR[11]. Le noyau **Panda** a été développé à la Vrije Universiteit Amsterdan et au MIT.

Le support à la multiprogrammation légère offerte par Panda est basé sur l'interface POSIX *threads* et fournit des primitives pour la création, la terminaison, la synchronisation (*join*, verrous, variables de condition) et pour la gestion des priorités. Panda implante ses *threads* de deux façons différentes. La première consiste à utiliser, si disponible, le noyau de *threads* du système cible. Dans ce cas là son implantation est réduite à un carrossage des primitives existantes sur ce noyau pour les adapter à l'interface Panda. C'est le cas de Panda sur Solaris. Dans le cas où le système cible ne présente pas de noyau de *threads*, ou s'il n'est pas adéquat (non efficace, manque de fonctionnalités, etc.), Panda offre son propre noyau implanté selon un modèle N :1. Panda offre des primitives de communication point à point, d'appel de procédures à distance et de communications collectives (groupe).

Le principe de base pour la réception de messages suit un modèle *upcall*. Les primitives point à point (*send*, *receive*) sont disponibles en versions synchrone ou asynchrone. Un message peut être reçu soit par une fonction enregistrée au préalable (primitive *register*) soit par une opération explicite de *receive*. Pour l'appel de procédure à distance, Panda propose une solution classique où un client demande l'exécution d'une procédure à un processus distant (*call*) et attend le retour (*reply*) d'un résultat. La procédure exécutée doit être enregistrée auparavant (primitive *register*).

La communication de groupe présente une sémantique d'ordre total qui garantit que, en cas d'envois multiples, les messages sont reçus dans un même ordre par tous

les processus appartenant au groupe impliqué. Les primitives de communication de groupe permettent à un processus de rejoindre un groupe (*join*), de quitter un groupe (*leave*), et d'envoyer un message au groupe (*send*).

La machine virtuelle **Panda** est composée d'un ensemble de processeurs virtuels, appelés *platforms*, selon la terminologie Panda. Chaque *platform* exécute un démon responsable de la réception et du traitement des messages. Chaque fois qu'un message arrive, le démon réalise un **upcall** vers le traitant de réception de messages de Panda. La détection de l'arrivée de messages est faite par **scrutation et par interruption** de façon intégrée. Le choix de l'une ou l'autre méthode est fait dynamiquement en fonction de l'état d'exécution de l'application. La méthode par défaut est celle de l'interruption. Si aucun *thread* n'est prêt à exécuter, Panda commute la détection par interruption vers celle par scrutation. Lorsqu'un *thread* devient actif, Panda repasse au mode interruption.

L'autre caractéristique importante de Panda est sa portabilité assuré par la couche *interface système*. Tous les détails spécifiques d'une architecture cible sont encapsulés par cette couche. Ainsi porter Panda sur une nouvelle architecture consiste essentiellement à (a) implanter la couche système pour cette architecture, et (b) à réaliser des réglages fins sur la couche Panda. Panda a été porté sur plusieurs systèmes d'exploitation (Amoeba, SunOs, Solaris et Parix) et machines (CM5, Parsytec GCel, Parsytec PowerXplorer, CS2 et grappe de monoprocesseurs SPARC reliés par myrinet).

L'environnement **Panda** a été employé comme base pour une série d'expérimentations. Plusieurs variantes d'implantations d'ORCA ont donné des résultats très intéressants sur l'intégration des techniques de scrutation et d'interruption [110] [111]. Ces expérimentations ont été conduites sur une grappe de monoprocesseurs (SPARC) reliés par un réseau myrinet. Le système d'exploitation utilisé a été Amoeba, et, comme protocole réseau, une version modifié de Fast Messages de façon à permettre l'activation et la désactivation des interruptions.

D'autres expériences ORCA ont été menées pour évaluer l'utilisation de ces méthodes et les surcoûts dûs aux changements de contexte entre les *threads* Panda et Orca [19][125]. Ici, la base expérimentale a été une grappe de monoprocesseurs SPARC, tournant sur Amoeba, interconnectés par un réseau ethernet à 10 Mbits/sec.

4.3.4 MPI-IBM

Le produit MPI disponible sur le machines SPx d'IBM est *thread-aware*. Le premier effort d'intégration des *threads* et des communications fait par IBM a été le prototype appelé MPI-F [79]. Ce prototype a été remplacé par la version MPI-IBM 2.3 disponible depuis la version AIX 4.2.

MPI-F est basé sur MPICH où toutes les couches plus basses ont été réécrites pour utiliser MPL, la bibliothèque de communication native des machines IBM SP1 et IBM SP2. MPI-F projette (mappe) la mémoire du sous-système de com-

munication directement dans la mémoire des processus utilisateurs. Cette approche évite les copies entre la mémoire d'un processus utilisateur et l'adaptateur de communications employé par les machines IBM SP. Par conséquent, les primitives de communication de MPI-F présentent un débit et une latence très proche de celle de la bibliothèque native MPL. L'autre composant de MPI-F est une bibliothèque de processus légers accessible par le sous-système de communication. Au niveau utilisateur, MPI-F offre une interface de programmation conforme à la norme POSIX (plutôt DCE Draft 4). L'implantation est en mode utilisateur (N :1).

Étant donné que le noyau de *thread* est de type N :1, MPI-F ne permet pas à un *thread* de se bloquer sur une opération de communication. Il est donc nécessaire de promouvoir des mécanismes non bloquants pour la progression des communications. Dans MPI-F l'avancement des communications se fait donc soit par **interruption** lors de l'arrivée d'un message, soit par l'appel à une fonction explicite de **scrutation**. L'interface de communication des machines IBM-SP (TB2) offre un mécanisme d'interruptions pour signaler l'arrivée d'un message. La gestion des interruptions introduit un tel surcoût sur la latence de message que ce mode est inutilisable. La scrutation explicite, ou périodique, est donc normalement utilisée. L'approche retenue par MPI-F, parmi celles possibles [111], est la suivante :

- une scrutation du réseau est faite chaque fois que l'application réalise un appel à une primitive de la bibliothèque de communication.
- l'ordonnanceur de *threads* du noyau réalise la scrutation à certaines phases de la procédure d'ordonnancement.
- la scrutation est déclenchée par un « chien de garde » qui s'occupe de scruter le réseau périodiquement par l'intermédiaire d'une interruption horloge ; cela garantit une fréquence minimale de scrutations lorsque les autres n'ont pas été réalisées ;

Le prototype MPI-F tournait sur les versions AIX 3.2. Le produit MPI-IBM 2.3 est disponible depuis la version AIX 4.2. La différence fondamentale entre les deux systèmes AIX est la présence des *threads* système (modèle 1 :1). Cependant, le problème de détermination du moment où un message est disponible sur l'interface réseau reste. Essentiellement trois modes différents de scrutation sont proposés : ***poll***, ***yield*** et ***sleep***.

Dans le premier mode, ***poll***, le *thread* qui effectue l'appel de communication bloquant, réalise une attente active (*busy wait*) de la fin de l'opération. Le désavantage de cette méthode est le gaspillage du temps de calcul au détriment des autres *threads*. En effet ce *thread* utilise tout le *quantum* de temps qui lui est dédié pour réaliser l'attente avant de céder le contrôle. Dans le mode ***yield***, le comportement est légèrement modifié, le *thread* assure toujours la scrutation mais « passe la main » (*yield*) si l'opération n'est pas faite. Dans le troisième mode, ***sleep***, le *thread* qui réalise un appel bloquant est suspendu (*blocked*). Chaque fois qu'un message destiné à MPI arrive à l'interface réseau, MPI recherche le *thread* qui correspond au message arrivé et le débloque (passage à un état *ready*).

Dans [32] une évaluation de ces différentes politiques est faite. Ces tests portent sur une grappe de monoprocesseurs. Dans le cadre spécifique de cette étude, le mode *yield* a donné les meilleurs résultats. Toutefois ces modes de fonctionnement présentent une différence au niveau efficacité selon le rapport de calculs et de communications et le nombre de *threads* bloqués à l'attente d'événements de communication. La meilleure politique ne peut être determinée que relativement à une application donnée.

4.3.5 PM^2

Le noyau exécutif **PM^2**[120], acronyme pour *Parallel Multithread Machine*, est un environnement de programmation parallèle portable conçu pour être le noyau commun aux différentes couches logielles du projet ESPACE[8]. Ce projet propose un modèle de programmation et d'exécution pour la réalisation d'applications parallèles irrégulières sur des architectures distribuées. L'idée de base de ce projet est d'atteindre une « virtualisation totale » des architectures sous-jacentes pour supporter de façon efficace et transparente l'exécution d'une application sur un système distribué quelconque.

Le modèle de programmation **PM^2** est basé sur un découpage de l'application en un ensemble de tâches. Le mécanisme essentiel pour ce découpage est celui de l'**appel léger de procédure à distance** (*Lightweight Remote Procedure Call* - LRPC). Il consiste à créer sur le nœud distant, un *thread* spécifique pour la réalisation d'une procédure. Ces procédures sont appelées **services**. Ceci correspond au modèle **popup** pour le traitement de messages. L'environnement **PM^2** offre trois variantes de l'appel léger de procédure à distance (LRPC) :

Appel synchrone : un appel léger de procédure à distance synchrone correspond à la sémantique de l'appel de procédure à distance « classique » ; c'est à dire que le *thread* qui l'effectue est bloqué jusqu'à l'arrivée du résultat.

Appel à attente différée : un appel léger de procédure à distance à attente différée est réalisé en deux temps. Initialement, le *thread* effectue l'appel à une procédure à distance, et obtient en retour immédiat un descripteur de l'appel ou *clé*. Le *thread* peut attendre et accéder au résultat du service par l'intermède de la *clé*.

Appel asynchrone : Dans un appel léger de procédure à distance asynchrone aucun retour de résultat n'est attendu par l'appelant.

Il existe un autre mode pour l'exécution de la procédure distante : le mode **quick**. Dans ce mode aucun *thread* n'est créé sur le nœud distant pour exécuter le service. L'exécution est prise en charge directement par le *thread* responsable de l'extraction du message du réseau. En d'autres termes, le mode « quick » im-

[8] « Exécution Support for Parallel Applications in high performance Computing Environments »

plante un modèle **message actif**. Par conséquent, il impose les mêmes restrictions aux appels effectués par un service (interdiction d'appels bloquants).

Les concepteurs de **PM²** ont décidé de développer leur propre noyau de processus légers dans le but d'avoir la garantie d'un même comportement sur les différentes plates-formes. Le noyau de processus légers MARCEL [116] répresente le résultat de cet effort. MARCEL implante un noyau de *threads* en espace utilisateur (modèle N :1) avec un ordonnancement préemptif à priorité basé sur une seule liste de processus prêts à exécuter. Chaque processus léger dispose d'un nombre de *quanta* de temps processeur proportionnel à sa priorité. Cet ordonnancement garantit qu'un *thread* x recevra un pourcentage de temps processeur donné par :

$$P_{proc} = \frac{\text{Priorité(x)}}{\sum_{i=0}^{n} \text{Priorité}_i\text{(i)}} \qquad i \in \{\text{processus légers prêts}\}$$

Le noyau MARCEL a été porté sur plusieurs architectures processeurs (Alpha, sparc, intel, power PC et Mips) et plusieurs systèmes d'exploitation (OSF, Unicos, solaris, SunOs, linux, AIX, IRIX). MARCEL est compatible avec la norme POSIX, et intègre certaines extensions. Parmi ces extensions, les plus importantes sont le support à la migration des processus légers et la politique d'ordonnancement.

PM² utilisait initialement le noyau de communication PVM. Afin de profiter des nouveaux réseaux haut débit comme Myrinet[24] et SCI[99], des interfaces efficaces BIP[132],VIA[62], un noyau de communication léger a été développé : MADELEINE [26]. **MADELEINE** [26] est une interface de communication née de la constatation qu'un mécanisme de type message actif ou *upcall* est le mécanisme de base d'une bibliothèque de communication efficace. Cette interface n'a pas été conçue pour être utilisée directement pour une application utilisateur, mais pour supporter le développement d'environnements de programmation parallèle distribuée basée sur *threads* du type **PM²**. L'intégration de MARCEL et MADELEINE donne **PM² High-Perf**.

Pour atteindre les objectifs d'efficacité et de portabilité, MADELEINE est composée de deux niveaux. Le premier constitue une interface de programmation à partir de laquelle il est possible de construire des primitives de communication basées sur l'appel de procédure à distance. Cette interface offre un ensemble de primitives simples pour l'envoi et la réception de messages, ainsi que pour l'emballage et le déballage de données. À ce niveau, MADELEINE considère un message comme une séquence d'une ou de plusieurs zones contiguës de mémoire localisées dans l'espace d'adressage utilisateur. Les zones mémoires sont emballées et ensuite envoyées vers un processus destinataire. La seule restriction est que du côté destinataire le déballage soit fait dans un ordre cohérent à celui de l'emballage.

L'implantation de MADELEINE est *thread aware* et repose sur une couche d'adaptation à un réseau. Cette couche suit un modèle de type **message actif**. Sa définition est faite pour profiter au maximum des possibilités de « zéro-copie » des interfaces réseaux actuels. MADELEINE offre une primitive spécifique (*poll*), c'est

à dire que la méthode de scrutation est « par demande ». MADELEINE présente aussi un mode de fonctionnement intégré au noyau MARCEL. Dans ce cas, la scrutation est assurée par l'ordonnanceur de MARCEL à la façon de « scheduller polls » de **Chant**. A l'heure actuelle MADELEINE est disponible sur les systèmes suivants : VIA [62] , BIP [132], SBP [25], Dolphin[9] SCI [99] , TCP/IP [148][149], PVM [85] et MPI [147].

PM^2 High-Perf est disponible sur des grappes de monoprocesseurs. Un travail en cours a pour objectif de le porter sur des SMP. Ce travail est essentiellement le développement d'une version SMP de MARCEL[10].

4.4 ATHAPASCAN-0

Nous avons vu dans les sections précédentes plusieurs projets de recherche d'intégration de la multiprogrammation légère et des communications de façon portable et efficace. Le noyau exécutif **ATHAPASCAN-0** a été conçu avec ce même objectif à la même période et a fait face aux mêmes problèmes. L'évolution et l'amélioration de ce noyau exécutif étant le sujet de ce travail de thèse, nous allons le décrire plus précisément que les autres projets.

4.4.1 Le modèle de programmation et les fonctionnalités d'ATHAPASCAN-0

ATHAPASCAN-0 propose d'étendre le modèle de réseau statique de processus « lourds » communicants, fourni par le noyau de programmation MPI, à celui de **réseau dynamique de *threads* communicants**. Un programme ATHAPASCAN-0 s'exécute sur une machine parallèle virtuelle composée d'une ou plusieurs **tâches** en utilisant un modèle SPMD[11]. Les **tâches ATHAPASCAN-0** correspondent à des processus « lourds » créés lors du lancement de l'application sur des machines physiques (nœuds). Plusieurs tâches peuvent être placées sur un même nœud. Le nombre total de tâches qui composent l'application est défini au moment du lancement de l'application et reste immuable jusqu'à la terminaison du programme[12]. Leur identification est basée sur une numérotation consécutive à partir de zéro.

Une tâche est le support d'exécution de *threads*. Un *thread* peut créer des ***threads* localement ou à la distance** (sur d'autres tâches). Les *threads* locaux - ou **esclave** selon la terminologie ATHAPASCAN-0 - puisqu'ils existent sur un même espace

[9] http ://www.dolphinics.no

[10] On attend avec impatience le prénom de ce nouveau bébé.

[11] Single Program Multiple Data.

[12] En fait, la création et le placement des tâches ATHAPASCAN-0 sur les nœuds physiques ne font pas partie de l'environnement de programmation proposé par ATHAPASCAN ; mais du support d'exécution offert par le MPI de la machine cible.

d'adressage, communiquent entre eux par la mémoire partagée. La synchronisation est faite par les mécanismes classiques de verrous, sémaphores et variables de conditions offertes par les primitives ATHAPASCAN-0. La création de *threads* à distance est réalise par le concept de **service**. Pour ces *threads* la synchronisation et le partage de données sont faits explicitement par l'échange de messages.

Un service est une procédure appelable à distance. Un service est identifié par un numero, et il est déclenché par une « requête de service urgent » ou par une « requête de service ». Dans le premier cas la procédure associée est exécutée selon le principe *upcall*. Dans l'autre cas l'exécution de la procédure (service) est assurée par la création d'un *thread* dédie à cette fin (*popup*). Dans les deux cas, un message est transmis en paramètre dans un tampon alloué exclusivement pour cela.

Les *threads* peuvent établir un nombre quelconque de liaisons point à point identifiées par un numéro : le **port**[13]. Les opérateurs d'émission et de réception existent en version bloquante et non bloquante. Pour ce dernier cas, une requête est associée à la communication et permet de tester ou attendre sa terminaison. Toutes les communications nécessitent l'emballage et le déballage de données dans une zone mémoire (tampon). Ceci est pris en charge directement par les opérateurs de communication pour toutes les données ayant une organisation contiguë et régulière en mémoire (séquence d'entiers, flottants, etc). Dans le cas contraire il faut explicitement emballer et déballer les données dans un tampon. Les primitives ATHAPASCAN-0 prévues à cette fin suivent la philosophie MPI.

4.4.2 La réalisation d'ATHAPASCAN-0

Le noyau exécutif **ATHAPASCAN-0** a pour but l'intégration de la multiprogrammation légère et de la communication. Le choix d'intégration est le plus simple possible et juxtapose les concepts propres à la multiprogrammation légère et à la communication. La réalisation d'ATHAPASCAN-0 est conforme au modèle général précédemment décrit. **ATHAPASCAN-0** est structuré en deux couches. La couche supérieure, **l'interface de programmation ATHAPASCAN-0**, est celle qui offre aux utilisateurs les fonctionnalités mentionnées à la section précédente pour le développement d'applications ATHAPASCAN-0. La couche inférieure, **Akernel**, réalise l'intégration de la bibliothèque de communication MPI avec une bibliothèque de *threads* (fig 4.7). C'est l'**interface de portabilité** d'**ATHAPASCAN-0**. Les primitives offertes par cette interface sont celles des standard POSIX Threads et MPI. Ces dernières sont rendues *thread-aware* selon la technique décrite en 4.2.2.

La sous-couche nommée API-MPI définit un sous ensemble des fonctions MPI de communications point à point en protégeant leurs accès par un verrou, si la bibliothèque MPI est *thread-unsafe* (pour une version MPI *thread-aware*, cette protection n'est pas faite). La sous-couche API-Threads homogénéise les différentes interfaces des divers noyaux de *threads* par rapport aux noms des fonctions, le nombre

[13] un port correspond à un communicateur et une étiquette MPI.

de paramètres, leur ordre, etc, vers une interface propre à Akernel.

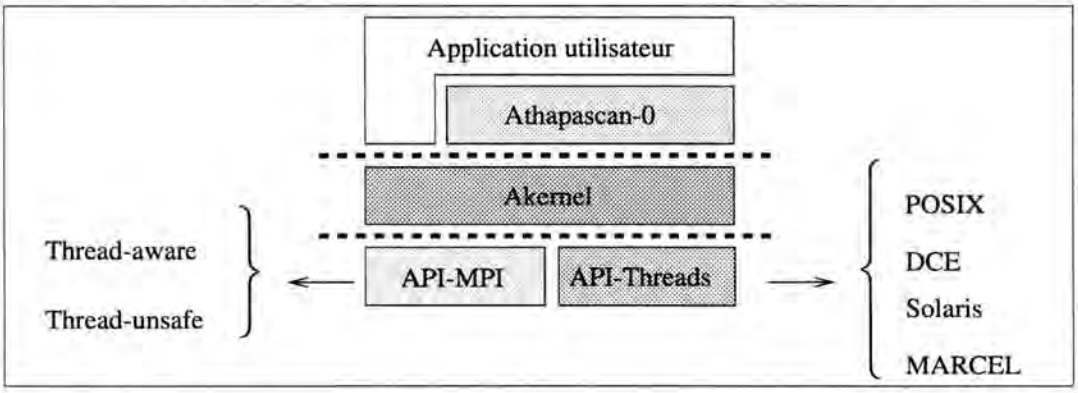

Figure 4.7 *L'architecture en couches d'*ATHAPASCAN-*0*

ATHAPASCAN-0 offre comme fonctionnalités supplementaires le lancement d'activités à distance (services) selon les modèles *upcall* et *popup*. Il utilise pour cela les communications bi-point offertes par la couche Akernel. Toutes les requêtes de services sur un nœud sont traduites en une émission d'un message à destination de ce nœud contenant la description de la requête. Selon le type de service, urgent ou non, ces messages sont émis vers un de deux ports prédéfinis reservés à ATHAPASCAN-0. Deux démons sont responsables de l'exécution des services, un démon pour chaque type de service. Chacun des deux démons est en attente de message sur un des ports réservés. À la reception d'un message de « requête de service urgent », le démon de service urgent identifie le service démandé et exécute la procédure correspondante dans sa propre pile d'exécution. C'est un modèle *upcall*. Dans le cas contraire, d'une « requête de service », le démon de service non urgent après avoir identifié le service demandé crée un nouveau *thread* pour l'exécuter. Ceci correspond à un modèle *popup*.

La couche Akernel assure l'intégration du noyau de *thread* et de la partie communication point à point de MPI. Son principal travail est de rendre cette partie de MPI *thread-aware*, c'est à dire de gérer les accès concurrents des communications et d'assurer la progression correcte des communications (cf. section 4.2.1). **Akernel** offre à partir des implantations non *thread-safe* de MPI, un ensemble de primitives de communications *thread-aware* en « encapsulant » les primitives critiques de MPI par un verrouillage de protection. Ainsi les primitives sont exécutées en exclusion mutuelle par un seul *thread* à la fois. Cependant les primitives potentiellement bloquantes ne peuvent pas suivre cette voie car on risque de bloquer MPI pour des durées trop importantes ou d'engendrer des interblocages.

La solution adoptée par **Akernel** pour gérer les accès concurrents à MPI suit le schéma présenté à la section 4.2.1. En effet MPI présente des versions non bloquantes de touts les opérateurs de communication bi-point. Ainsi lorsqu'un *thread* réalise une primitive de communication le niveau Akernel initie en exclusion mutuelle l'opération MPI non bloquante correspondante (fig. 4.8). Un descripteur de

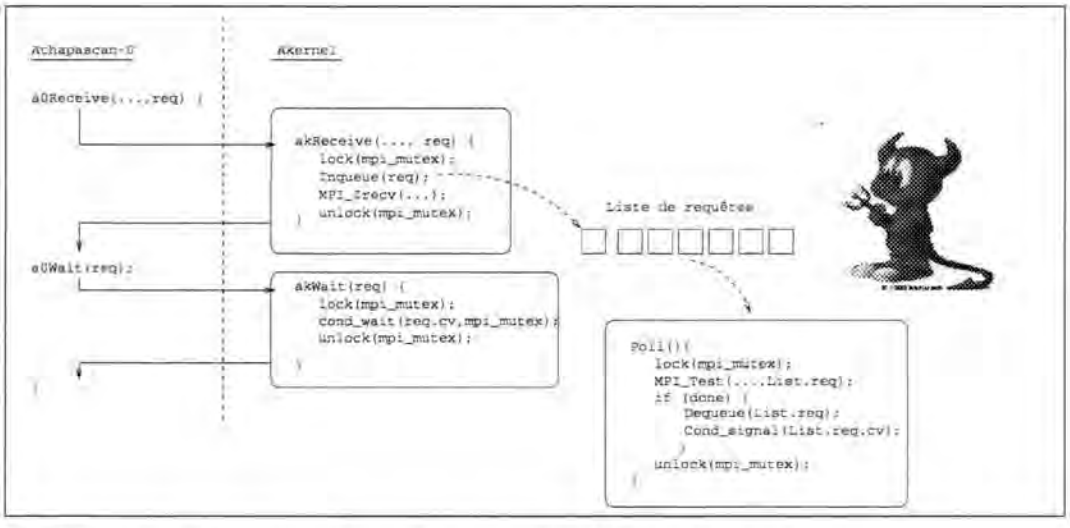

Figure 4.8 *L'algorithme Akernel pour rendre MPI* thread-aware

requête y est associé. Ce descripteur associe le descripteur de l'opération non bloquante MPI et une condition POSIX. Ainsi le *thread* peut venir tester ou attendre la terminaison du transfert asynchrone.

Le second problème est d'assurer la progression et la détection de terminaison des opérateurs asynchrones. Nous avons écarté la méthode basée sur une interruption car généralement cette interruption n'est pas disponible de façon efficace au niveau applicatif. En conséquence, **Akernel** emploie un *thread* spécialisé, le **démon de communication**, pour réaliser périodiquement la **scrutation** c'est à dire appeler MPI pour assurer l'avancement des communications et trouver les opérations terminées.

L'activité du démon de communication va dépendre totalement de l'ordonnanceur disponible sur le noyau de *threads* utilisé. La solution adoptée par les développeurs de la version initiale d'ATHAPASCAN-0 a été de fournir différents modes d'activation du démon. Le choix du mode de fonctionnement est fait en fonctions des caractéristiques du système cible. Les modes d'activation du démon proposés sont les suivants :

- **Mode 1 :** Il est destiné à un noyau de *threads* à priorité fixe, le démon tourne avec la priorité maximale et se suspend lorsqu'il a terminé une étape de scrutation. Il doit être débloqué :
  - par le dépôt d'une nouvelle requête de communication.
  - par un *thread* « chien de garde » qui ne tourne que lorsqu'aucun *thread* n'est actif. On peut alors scruter en continu tant qu'aucun *thread* de calcul n'est prêt.
  - au bout d'un délai fixe de façon à garantir une période de scrutation minimale.

- **Mode 2 :** Il correspond à un noyau où celui-ci effectue un partage équitable du temps processeur (priorité dynamique à la UNIX, temps partagé), le démon effectue une étape de scrutation à chaque activation par l'ordonnanceur et « passe la main » (yield()). Si la fréquence de scrutation est trop importante on peut retarder l'activation d'un délai donné.
- **Mode 3 :** Il est destiné à un noyau où les *threads* tournent jusqu'à blocage explicite (stratégie FIFO sans priorité, par exemple), le démon effectue une étape de scrutation à chaque activation par l'ordonnanceur et « passe la main » (yield()).
- **Mode 4 :** cette situation correspond au cas où le noyau de communication est du type *thread aware*. Dans cette situation l'accès concurrent et l'avancement des communications sont garantis par l'implantation de la bibliothèque de communication. Ici la couche Akernel se réduit à une simple conversion (macros) de l'interface Akernel aux opérations MPI équivalentes.

Dans tous les cas, l'activité du démon est un parcours partiel ou complet de la liste de requêtes MPI pour en tester la terminaison. Le nombre de requêtes testées détermine le temps d'occupation processeur pour chaque période d'activation du démon. Le démon doit-il vérifier toutes les requêtes présentes dans la liste ? Ou une partie seulement pour borner son temps d'exécution ? La solution prise a été celle de parcourir la liste des requêtes jusqu'à en trouver une qui soit prête ou que la liste soit complètement parcourue. On voit ici l'importance et l'interdépendence de l'ordonnancement de l'étape élementaire de scrutation et les délais caractérisant la période d'activité. La détermination des bons choix exige toujours une étape importante de réglage.

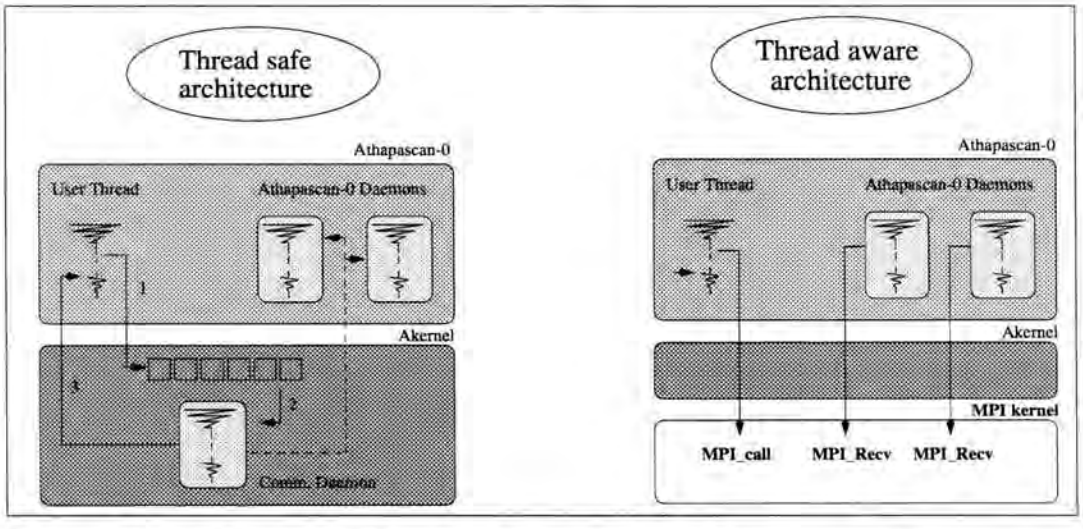

Figure 4.9 *Architectures de base pour la mise en œuvre des communications*

4.4.3 Le portage et le réglage d'ATHAPASCAN-0

L'organisation en couches d'ATHAPASCAN-0 illustré à la figure 4.7 nous a permis de réaliser des portages d'un système à l'autre assez rapidement, à la condition que l'architecture cible fournisse un noyau de *threads* et une implantation de MPI. Si, pour assurer sa portabilité, ATHAPASCAN-0 profite de la disponibilité et des fonctionnalités d'un noyau POSIX et de MPI, il hérite de toutes les caractéristiques et contraintes d'implantations sur une architecture et un système d'exploitation visé. C'est en particulier le cas pour, d'une part, le mode d'implantation du noyau (N :1, 1 :1, M :N) et, d'autre part, les politiques d'ordonnancement. Ainsi le comportement d'ATHAPASCAN-0 peut changer si le noyau de base est préemptif ou non, si la politique d'ordonnancement est FIFO ou temps partagé, si les *threads* sont au niveau système ou utilisateur, etc. À chaque portage ces variantes imposent une importante étape de « réglage » pour obtenir les meilleures performances sur une architecture cible. Le tableau 4.1 donne des exemples de portage.

Tableau 4.1 *Exemples de portages d'*ATHAPASCAN-0
Obs. : Les modes font référence à ceux présentés à la fin de la section 4.4.2

Système d'exploitation	Noyau de de *threads*	Noyau de communication	Mode de Activation	Ordonnanceur
AIX 3.2	DCE Threads D-4 (modèle N :1)	MPI-F	Mode 1	FIFO à priorités fixes
AIX 4.2	POSIX Threads (modèle 1 :1)	MPI-IBM 2.3	Mode 4	Temps partagé, priorités dynamiques
AIX 4.2	POSIX Threads (modèle 1 :1)	LAM MPICH/MPL	Mode 2	Temps partagé, priorités dynamiques
Solaris 2.5/2.6	POSIX Threads (modèle N :1)	LAM	Mode 1	FIFO à priorité fixes
Solaris 2.5/2.6	POSIX Threads (modèle 1 :1)	LAM	Mode 2	Temps partagé, priorités dynamiques
Linux 2.x	Linux (POSIX) (modèle 1 :1)	LAM	Mode 2	Temps partagé, priorités dynamiques

Les portages du tableau 4.1 correspondent aux versions considérées « stables » par les développeurs d'ATHAPASCAN-0. D'autres portages ont été fait dans un cadre d'expérimentations générales (sur des grappes de monoprocesseurs) :

- Noyau de *threads* : MARCEL, MIT threads (Provenzano), SGI IRIX 6.x POSIX *threads*, OSF 4.x POSIX *threads*, HP-UX 10.x DCE *threads*. Les divers noyaux POSIX diffèrent par certaines fonctionnalités, notamment celles liées à l'ordonnanceur et traitement de priorités.
- Noyau de communication : MPICH, T3E MPI (une version de CHIMP), et MPI-BIP.

En principe, les systèmes récents offrent une bibliothèque MPI *thread-aware*, comme par exemple le système AIX 4.2 (MPI-IBM 2.3), la couche **Akernel** perd son utilité (fig. 4.9.b). Sa fonction est réduite à l'homogénéïsation de l'interface des *threads*. Cependant, notre expérience d'utilisation d'un noyau MPI de ce type nous a donné des résultats moins efficaces que notre solution [32]. Par conséquent nous avons décidé de garder notre solution avec *démon de communication* (Mode 2).

4.5 Bilan

Il est facile de tirer un bilan de l'état de l'art d'intégration de la communication à la multiprogrammation légère. Toutes les expériences ont convergé vers un ensemble limité de principes (message actif, *upcall*, *popup*) et de techniques de scrutation par interruption, par démon spécialisé ou intégrées à l'ordonnanceur. Les différents projets ont été confrontés au même problème : celui d'assurer la progression optimale des communications sans dégrader le service de calcul. L'origine du problème se trouve dans un couplage insuffisant de l'ordonnancement des *threads* et de l'interface de communication. L'effet de cette lacune est accru par le comportement inconnu des ordonnanceurs disponibles. Tous les projets ont nécessité des phases de réglages systématiques à chaque portage sur une nouvelle machine[14].

On remarquera que toutes les expériences ont porté sur des grappes de monoprocesseurs. On peut donc se poser les questions suivantes :

- Les principes développés sur des grappes de monoprocesseurs sont ils transférables sur des grappes SMP ?
- Peut on profiter du parallélisme SMP pour améliorer le recouvrement calcul communication ? Est il plus intéressant d'utiliser un des processeurs supplémentaires pour la communication que pour le calcul ?
- La mise en œuvre de ces principes et techniques pose-t-elle des problèmes spécifiques du fait d'un **parallélisme réel** entre *threads* (contention sur les caches, les verrous, etc) ?
- L'ordonnancement des *threads* sur SMP va-t-il aggraver la difficulté du contrôle de la scrutation et de la progression des communications ? pourra-t-on trouver un réglage approprié et quels en sont les paramètres critiques ?

Essayer de répondre à ces questions est l'objet du travail de thèse présenté aux chapitres suivants. Le chapitre 5 est dévolu au portage d'ATHAPASCAN-0 sur SMP et à la résolution spécifique des problèmes de programmation sur SMP. Le chapitre 6 se préoccupe de l'évaluation des performances offertes par une telle implantation.

[14] un simple accroissement de puissance des processeurs suffit à déregler.

5

ATHAPASCAN-0 sur multiprocesseurs

« Le trop d'expédients peut gâter une affaire, on perd du temps au choix, on tente, on veut tout faire : n'en ayons qu'un ; mais qu'il soit bon. » Jean de La Fontaine, Le chat et le renard

Ce chapitre présente donc une partie de notre travail de thèse. Il concerne le portage d'ATHAPASCAN-0 sur multiprocesseurs. Il porte essentiellement sur les difficultés introduites par le parallélisme réel et la synchronisation de *threads*. La première section est dévolue à un rappel des problèmes spécifiques liés à la synchronisation entre *threads* s'exécutant sur des processeurs différents et partageant de la mémoire. La section suivante présente les modifications d'ATHAPASCAN-0 nécessaires à une exécution efficace sur SMP. Cette version est appelée ATHAPASCAN-SMP. Il termine par une simple caractérisation du surcoût introduit par ATHAPASCAN-SMP par rapport au noyau de *thread* de base.

5.1 Les multiprocesseurs

Les problèmes posés par l'architecture des multiprocesseurs et les sources d'inefficacité correspondantes sont aujourd'hui bien connus. Ils sont plus ou moins graves selon la taille du multiprocesseur et la technologie d'implantation du réseau d'interconnexion mémoire. On peut considérer que les problèmes suivants sont critiques pour les multiprocesseurs de grande taille et grande puissance (SGI Origin 2000, SUN Enterprise 10000, HP Exemplar, etc) ou pour les multiprocesseurs de faible taille à faible coût (cartes multiprocesseur pentium, UltraSparc, DEC, etc) :

- écroulement due à un taux d'accès concurrent à la mémoire trop important.
- écroulement due à un taux de conflit de synchronisation trop important.

Dans les sections suivantes nous allons détailler chacun de ces deux aspects.

5.1.1 La mémoire cache

La caractéristique principale d'un multiprocesseur est le partage de la mémoire entre les processeurs. Quel que soit le temps d'accès des processeurs aux zones mémoire (uniforme ou non) celui-ci est trop important pour la technologie actuelle de microprocesseurs. Il est nécessaire d'utiliser des mémoires caches pour masquer ces trop fortes latences. Le fait que les processeurs puissent lire et modifier en parallèle les données en mémoire impose de maintenir cohérentes les copies de ces données se trouvant éventuellement dans les caches. L'architecture actuelle des caches est la suivante :

- Les données sont amenées en cache sur la base d'une ligne de cache qui typiquement est un multiple (puissance de 2) des valeurs élémentaires manipulables (entiers, flottants) par le processeur. Lorsqu'un processeur accède à une donnée et que celle-ci n'est pas trouvée dans le cache, il doit amener la ligne contenant la donnée depuis la mémoire (chargement). Cette opération peut exiger de sauver en mémoire une ligne se trouvant dans le cache si celui-ci est plein (remplacement).
- Lorsqu'une écriture de donnée est faite dans une ligne de cache il faut éventuellement propager cette modification aux autres caches. Ceci se fait soit en diffusant la nouvelle valeur (*write update*), soit en écrivant en mémoire la ligne modifiée et en diffusant l'indication de cette modification (*write invalidate*). Les caches se remettent à jour à partir de la nouvelle valeur de la ligne en mémoire.

On conçoit que le parallélisme d'un multiprocesseur va se traduire par une surcharge sur l'accès à la mémoire dû aux accès concurrents non conflictuels à la mémoire (chargement-replacement) et aux accès conflictuels à la mémoire. Sur le premier point, la solution « classique » est d'améliorer la localité des données manipulées par un processeur. Du point de vue du parallélisme, il faut éviter une situation dite de **faux partage** où une ligne de cache regroupe des données indépendantes de 2 processeurs au moins. En effet, dans ce cas toute écriture d'un processeur provoque une mise en cohérence inutile de l'autre processeur.

Le second point consiste à programmer en minimisant les conflits sur les variables partagées. C'est un problème de conception d'algorithme parallèle et de coordination des accès. Une approche matérielle est de ne faire la mise en cohérence que lors d'un point de synchronisation (*entry/release consistency*) . L'approche logicielle consiste à limiter par la synchronisation le nombre de processus en conflit (contrôle du degré de partage, exclusion mutuelle, etc).

5.1.2 La synchronisation

Dans un programme parallèle, la synchronisation est donc un moyen utilisé pour coordonner les activités concurrentes susceptibles d'accéder à des données partagées. Le mécanisme de synchronisation de base est celui de l'exclusion mutuelle (verrou), où on doit garantir qu'un seul *thread*, à un moment donné, exécute une portion du code (région critique). Il y a un coût associé à l'utilisation des verrous. Ce coût provient du coût des opérateurs eux même du fait de la complexité d'implantation mais aussi de la synchronisation réalisée qui réduit le parallélisme.

Implantation de verrous

Les mécanismes de synchronisation reposent sur deux opérateurs : la prise d'un verrou (*lock*) et la relâche d'un verrou (*unlock*). On peut voir un verrou comme une structure de données comportant :

- un booléen donnant l'état du verrou (pris, libre) ;
- une file enregistrant les *threads* bloqués en attente de la libération du verrou.

Les algorithmes 5.1 et 5.2 donnent le principe général de ces deux opérations. On constate deux choses. La première est que le coût de ces algorithmes est important dès que le verrou est pris car cela provoque un appel à l'ordonnanceur (pas 3, algorithme 5.1 et pas 2, algorithme 5.2) et une commutation de contexte d'un *thread* (sauvegarde) à un autre (restauration). Le second point est que la mise en œuvre des verrous suppose l'existence d'un mécanisme de verrouillage plus élémentaire permettant de modifier le verrou et la file associée de façon atomique afin d'éviter les incohérences possibles si deux processus (*threads*) tentaient d'exécuter ces procédures en même temps.

Algorithme 5.1 Lock(Mutex M)

```
1: while M.state ≠ free do
2:     Insert(M.queue, CurrentThread()) ;
3:     ThreadScheduler() ;
4: end while
5: M.State ← busy
```

Algorithme 5.2 Unlock(Mutex M)

```
1: if M.queue ≠ Empty then
2:     WakeUp(M.queue) ;
3: end if
4: M.State ← free
```

Programme 5.1 Primitive lock en assembly (SPARC)

```
1:  try : ldstub adress->register
2:        compare register, 0
3:        branch_equal done
4:        call go_to_sleep
5:        jump try
6:  done : return
```

La base pour une implantation de ce verrou primitif est une instruction de type *test & set* (T&S) c'est à dire un opérateur de lecture et modification atomique de la mémoire qui permet de réaliser l'exclusion mutuelle entre deux processeurs. Par exemple, sur le microprocesseur SPARC, cette instruction est la *ldstub* (*load and store unsigned byte*) laquelle réalise la lecture d'une position mémoire (octet) vers un registre et au même temps attribue la valeur 1 à cette position mémoire. Le programme 5.1 montre une implantation possible de la primitive *lock* en utilisant cette instruction.

Le tableau 5.1 donne un ordre de grandeur du coût des opérations *lock/unlock* (POSIX threads) dans le cas où le verrou est disponible. Nous observons que les coûts de ces primitives, comparés à certaines opérations élémentaires, sont assez importants, d'où l'intérêt d'en diminuer la fréquence d'utilisation.

Tableau 5.1 *Exemples de temps d'exécution pour certaines primitives élémentaires* (Caiapo : solaris 2.6, pentium @133Mhz ; Guarani : solaris 2.6, bi-pentium II @333Mhz ; Comanche : solaris 2.6, quadri-pentium pro @200Mhz ; Huron : solaris 2.6, bi-sparc @450Mhz)

Primitive	Caiapo	Guarani	Huron	Comanche
lock/unlock (POSIX threads)	0.754μs	0.347μs	0.281μs	0.570μs
appel à procédure(void)	0.113μs	0.034μs	0.058μs	0.058μs
coût par paramètre	0.013μs	0.003μs	0.007μs	0.007μs
itération boucle vide	0.045μs	0.018μs	0.042μs	0.030μs

L'utilisation de verrous

La façon d'utiliser les verrous est fortement dépendante d'un algorithme. Par exemple, au lieu de prendre et de relâcher un verrou plusieurs fois au cours d'un calcul, on peut le prendre une seule fois, réaliser tout le calcul, et le relâcher à la fin. En d'autres termes, on accroît la granularité du calcul qui ne peut être parallélisé.

L'emploi d'une granularité de verrouillage à gros grain diminue donc le coût de la réalisation des opérations de synchronisation, mais augmente la probabilité de **contention**, c'est à dire du nombre de *threads* qui risquent de se bloquer en attente

de cette ressource (verrou). Ceci implique des coûts supplémentaires liés à la procédure d'ordonnancement. Un autre désavantage est la réduction du parallélisme et la sous utilisation du multiprocesseur. Il est donc nécessaire de trouver un bon compromis entre la granularité (contention) et le nombre d'opérations de synchronisation. Les méthodes suivantes vont chercher à limiter les conflits soit en partitionnant les accès aux données soit en détectant des situations sans conflit :

- Des primitives de plus haut niveau comme les verrous de type *read-write*. Ici, plusieurs « lecteurs » sont autorisés à lire une donnée, mais un seul « écrivain » peut la modifier. Durant une opération d'écriture, aucune lecture n'est permise. Par contre, les lectures peuvent avoir lieu en parallèle.
- La partition des données partagées en plusieurs sous groupes, chacun protégé par un verrou différent. Un façon de procéder [142] est de débuter avec un nombre faible de verrous de granularité à gros grain et, selon la performance obtenue, réduire la granularité des sections et rajouter des verrous.
- L'utilisation d'algorithmes « libre de blocages » (*lock-free*). Ces algorithmes « optimistes » sont basés sur l'hypothèse que les conflits d'accès sont rares et qu'il n'est pas nécessaire de toujours synchroniser a priori (méthode pessimiste). On ne fait donc pas de contrôle a priori. Par contre, on doit savoir a posteriori (à la fin des accès) s'il y a eu conflit. Dans ce cas, on déclenche une procédure de correction qui est en général coûteuse. L'intérêt d'employer cette technique dépend donc du taux de conflit d'accès.

Si l'accroissement du grain n'améliore pas l'efficacité du fait de l'augmentation du nombre de threads à bloquer, il existe une technique de verrouillage spécifique aux SMP plus appropriée à la réalisation des exclusions mutuelles à grain fin. Cette technique est connue sous le nom de verrou à **attente active** (*spin lock*).

Attente active (*spin lock*)

Sur une machine monoprocesseur, bloquer un *thread* qui attend un verrou est logique car le processeur doit être libéré pour exécuter d'autres *threads*. Un seul, parmi ceux-ci, exécutera la libération du verrou attendu et débloquera le *thread* en attente de cette ressource. Sur une machine multiprocesseur, ce changement de contexte peut être évité en permettant au *thread* de tester de façon répétitive la disponibilité du verrou. C'est le verrou à **attente active** (*spin lock*). Pour que ceci marche, il faut garantir que le *thread* débloquant le verrou finisse par tourner. Ceci peut être garanti de deux façons. N'utiliser cette technique que pour des séquences où un verrou pris est libéré très rapidement sans blocage du *thread* propriétaire. Utiliser un mécanisme de temps partagé qui garantit que tous les *threads* tournent à tour de rôle.

On peut illustrer l'implantation d'un mécanisme de *spin_lock* à partir du programme 5.1 : il suffit d'ôter le pas 4. Ainsi un *thread* appelant la primitive *lock* (*spin*) s'exécutera jusqu'à qu'il obtienne le verrou ou que son temps d'exécution

soit épuisé (*quantum*). On notera dans ce deuxième cas un gaspillage du temps de calcul.

L'attente active devient intéressante lorsque la durée de l'exclusion mutuelle est voisine du temps de commutation de *threads*. Les mesures effectuées par Anderson [8] et Karlin [105] donnent une valeur de 50% et 100% du temps d'un changement de contexte. Il devient alors possible de mettre en place une opération de verrouillage qui effectuera une attente active pendant une durée équivalent à 50%-100% du temps de commutation. Si après une période d'attente active le verrou n'est pas devenu libre, un blocage classique est alors réalisé. On peut donc estimer la durée d'une section critique en deça de laquelle il est intéressant d'utiliser un verrou de type *spin lock*.

Le standard POSIX Threads met à disposition une primitive pour la réalisation de l'attente active : *pthread_mutex_trylock()*. Lors de son exécution, cette primitive essaie de prendre le verrou passé comme argument. En cas de réussite son fonctionnement équivaut à celui de la primitive *pthread_mutex_lock()*. Dans le cas contraire, elle retourne un code d'erreur qui peut être utilisé pour la réalisation de l'attente active. Le programme 5.2 donne un exemple d'implantation en langage C d'un verrou de type *spin lock* à l'aide des opérateurs POSIX. La boucle *for* (pas 3) a comme objectif de limiter le temps de réalisation de l'attente active ; la constante SPIN_COUNT est réglée de façon à fournir un temps d'exécution voisin de celui de commutation d'un *thread*.

Programme 5.2 Implantation de *spin_lock* avec des primitives POSIX (Solaris)

```
1:  void spin_lock(pthread_mutex_t *m) {
2:  int i;
3:    for(i=0; i<SPIN_COUNT; i++) {
4:      if (pthread_mutex_trylock(m) != EBUSY)
5:        return;
6:    }
7:  pthread_mutex_lock(m);
8:  }
```

Le tableau 5.2 donne les temps d'exécution de la primitive *pthread_mutex_trylock()* dans les cas de succès et d'échec, ainsi que le temps de changement de contexte[1] entre *threads* POSIX (*bound* et *unbound*). Les *threads bound* correspondent à un modèle 1 :1 et les *threads unbound* à celui de N :1.

L'utilisation intensive de l'attente active, par plusieurs *threads* sur un même verrou, peut cependant occasionner des problèmes de famine pour les *threads*. L'attente active est recommandée pour un nombre faible de *threads* [106].

[1]Ces valeurs sont ceux présentées à la fin du chapitre 2.

Tableau 5.2 *Temps d'exécution de* trylock *et de changements de contexte*
(Caiapo : solaris 2.6, pentium @133Mhz ; Guarani : solaris 2.6, bi-pentium II @333Mhz ; Comanche : solaris 2.6, quadri-pentium pro @200Mhz ; Huron : solaris 2.6, bi-sparc @450Mhz)

Primitive	Caiapo	Guarani	Huron	Comanche
trylock/unlock (POSIX threads)	0.775μs	0.340μs	0.320μs	0.570μs
trylock (verrou non disponible)	0.388μs	0.180μs	0.180μs	0.300μs
chang. contexte (bound)	66.32μs	12.54μs	14.64μs	17.72μs
chang. contexte (unbound)	12.74μs	3.690μs	6.05μs	6.24μs

Un autre inconvénient de l'attente active est lié au mécanisme de base (l'instruction *test & set*) offert par le microprocesseur pour l'implantation de primitives de synchronisation : *lock* et *spin_lock*. La boucle d'itération de la procédure *spin lock* (programme 5.2) est en fait une boucle *test & set*. L'exécution continue de *test & set* provoque une dégradation importante du fait que cette instruction demande un accès exclusif à une position de mémoire pour une lecture/écriture en court-circuitant les caches. Ceci provoque un écroulement de la voie d'accès (bus, liasion) au banc mémoire considéré. Il est possible de limiter cette perturbation en utilisant une variable d'état intermédiaire testée normalement en cache. Lorsque cette variable est mise à jour par le protocole de cohèrence de cache, on peut alors essayer d'acquérir le verrou de façon atomique. On trouvera dans le programme 5.3 une implantation portable d'un *spin lock* et en 5.4 une implantation exploitant une connaisance de l'implantation des verrous solaris 2.6 (donc non portable).

Pour démontrer cette effet nous avons donc exécuté les deux versions de programme (5.2 et 5.4) sur nos machines Solaris. La réalisation de ce test considère le verrou comme étant pris et l'exécution par un *thread bound* (modèle 1 :1). Le temps mesuré correspond au temps d'exécution de la boucle *for* interne à la fonction *spin lock*. On constate donc une différence d'environ 7 fois - pour les Intel - et de 3 fois - pour le SPARC - entre les deux versions ce qui démontre clairement le coût de l'instruction *test&set*.

Tableau 5.3 *Comparaison entre les algorithmes « conventionnel » (5.2) et « modifié » (5.4) pour l'attente active (boucle SPIN_LOCK=100)*
(Guarani : solaris 2.6, bi-pentium II @333Mhz ; Comanche : solaris 2.6, quadri-pentium pro @200Mhz ; Huron : solaris 2.6, bi-sparc @450Mhz)

Primitive	Guarani	Huron	Comanche
spinlock (trylock)	19.2μs	21.0μs	31.0μs
spinlock (modifié)	2.7μs	7.9μs	4.5μs

Programme 5.3 Implantation efficace de *spin_lock* avec des primitives POSIX (Solaris)

```
1:  typedef struct my_mutex {
2:   pthread_mutex_lock m ;
3:   int state ;
4:  }
5:
6:  void spin_lock(my_mutex_t *m) {
7:  int i ;
8:   for(i=0 ; i<SPIN_COUNT ; i++) {
9:     if (m->state) continue ;
10:    if (pthread_mutex_trylock(&(m->m)) != EBUSY) {
11:      m->state=1 ;
12:      return ;
13:    }
14:  }
15: pthread_mutex_lock(&(m->m)) ;
16: m->state=1 ;
17: }
18:
19: void spin_unlock(my_mutex_t *m) {
20: m->state=0 ;
21: pthread_mutex_unlock(&(m->m)) ;
22: }
```

Programme 5.4 Implantation non portable (exclusif Solaris 2.6)

```
1:  void spin_lock(pthread_mutex_t *m) {
2:  int i ;
3:   for(i=0 ; i<SPIN_COUNT ; i++) {
4:     if (m->__pthread_mutex_lock.__pthread_mutex_owner64==0)
5:       if (pthread_mutex_trylock(m) != EBUSY)
6:         return ;
7:   }
8:  pthread_mutex_lock(m) ;
9:  }
```

5.2 La version multiprocesseur d'ATHAPASCAN-0

Bien que conçu en vue d'être utilisé sur multiprocesseurs, ATHAPASCAN-0 comme ses homologues (Nexus, Chant, PM^2, etc), a été réalisé sur des machines monoprocesseurs. Le portage de cette première version sur un multiprocesseur a donné des résultats immédiats bien que décevants. Initialement le parallélisme réel du SMP a permis à un certain nombre de bogues de se manifester. Ensuite, les performances obtenues[2] ont été faibles du fait de l'importance des conflits de cache et de synchronisation (cf. section précédente).

Une série de modifications a donc été apportée et a conduit à la version ATHAPASCAN-SMP. Ces modifications ont porté essentiellement sur la résolution des conflits sur les verrous et la mise en place d'une méthode de scrutation tenant compte du parallélisme réel du SMP.

5.2.1 Adaptation du grain d'exclusion mutuelle

Comme décrit à la section 4.4.2 (fig. 4.8), MPI a été rendu *thread aware* de la façon suivante :

- Les *threads* appellent MPI pour soumettre une communication non bloquante. Il est nécessaire de faire ces opérations en exclusion.
- Les *threads* transmettent la requête associée à la communication au démon de scrutation. L'insertion et retrait dans la file de requêtes doit être fait en exclusion mutuelle.
- enfin le démon de scrutation parcourt la liste de requêtes (accès exclusif) pour tester si la requête est terminée (accès exclusif à MPI). Les *threads* se mettent en attente de terminaison via une condition POSIX associée à la requête. Cette condition est signalée par le démon de scrutation détectant la terminaison de la requête.

L'algorithme original présenté à la figure 4.8 est basé sur l'utilisation d'un seul et unique verrou (*mpi_mutex*) pour assurer l'atomicité de ces actions. Nous avons remarqué, dès les premiers tests, que lorsqu'on réalisait des applications avec un certain nombre de *threads* (3 à 4) communicants, le nombre de commutations de contexte était assez important[3] du fait d'un problème de contention sur un verrou. Conformément aux principes évoqués à la section précédente, nous avons cherché à réduire cette contention sur un verrou unique en partageant les données verrouillées et les sections critiques entre plusieurs verrous.

[2]Une fois les bogues corrigés.

[3]Ceci a été observé avec l'aide du système de fichier /proc de Solaris 2.6 qui enregistre un série de statistiques système.

Nous avons agi sur les trois points assurant le contrôle de MPI. Nous avons donc modifié l'accès exclusif à MPI, la gestion de la liste de requêtes et la synchronisation de fin des communications.

Accès exclusif à MPI

Le premier niveau de granularité que nous avons modifié est celui de la protection des appels à MPI pour le rendre *thread-safe*. Ceci a été fait par la création d'un verrou spécifique. Tous les appels MPI sont protégés par ce verrou. Tous les *threads* ATHAPASCAN-0 et le démon Akernel sont en compétition sur ce verrou. La durée de verrouillage varie en fonction de la primitive MPI réalisée. Le tableau 5.4 présente les primitives MPI utilisées à l'intérieur de la couche Akernel, et à titre indicatif leur temps d'exécution. Pour les primitives d'envoi et de réception de messages, la taille considérée pour les messages est nulle. Le surcoût d'ATHAPASCAN-0 est celui des primitives *trylock/unlock* (voir tableau 5.2).

Tableau 5.4 *Les primitives LAM-MPI et leur temps d'exécution (Solaris 2.6)*

	Pentium 133Mhz	Bi-PentiumII 333Mhz
MPI_Isend	259.60μs	90.96μs
MPI_Irecv	94.22μs	26.40μs
MPI_Iprobe	70.88μs	16.04μs
MPI_Cancel	0.564μs	0.135μs
MPI_Test	1.587μs	0.372μs

Synchronisation de fin de communication

Une variable de condition associée à une requête MPI permet à un *thread* ATHAPASCAN de tester ou d'attendre la fin de la communication non bloquante. Cette condition est signalée par le démon de communication. L'exclusion mutuelle nécessaire à l'emploi des variables de condition est assurée aussi par le verrou unique *mpi_mutex* (fig. 4.8).

Nous avons introduit un *verrou* par requête de communication. Cette solution n'implique aucune contention sur ce verrou car il est privé à la requête. En effet, ce verrou est nécessaire uniquement à cause de la sémantique de fonctionnement des variables de condition (section 2.4.4). Un conflit d'accès ne peut se produire qu'entre le *thread* communicant et le démon de scrutation. Ce conflit est très rare car le démon et le *thread* communicant doivent s'exécuter simultanément, un pour acquérir le verrou pour signaler la fin de communication (le démon) et l'autre pour attendre la fin de la communication (le *thread*).

Par rapport à la solution initiale d'ATHAPASCAN-0 monoprocesseur, le coût d'exécution est le même. La différence est au niveau de l'occupation mémoire. Si l'on considère qu'un verrou consomme v octets en mémoire et une variable de condition c octets, la solution originale représente un coût de $(v + n \times c)$ octets contre $n \times (c + v)$ octets, où n est le nombre de requêtes. Par exemple, dans l'implantation POSIX threads sur solaris 2.6 (intel), pour une liste de requêtes de 1024 élements, le surcoût mémoire représente 24552 octets ($c = 16$ et $v = 24$).

La gestion des requêtes

Le cycle de vie d'une requête et le parcours correspondant dans les listes de gestion sont décrites dans la figure 5.1. Initialement une requête est libre et enregistrée dans la liste de requêtes libres. Elle est acquise par un *thread* initiant une communication non bloquante. Après initiation, elle est inserée dans la liste des requêtes en cours. Cette liste est parcourue par le démon. Lorsqu'il détecte une terminaison de requête il la retire de cette liste. La requête est dite « terminée ».

Le *thread* initiateur d'une communication revient tester ou attendre la fin de communication par l'intermédiaire de la variable de condition associée à la requête. S'il la trouve « terminée » il l'insère dans la liste de requêtes libérées. Dans le cas contraire, le *thread* est bloqué sur cette condition et le démon lors de la terminaison de la requête le lui signalera. Dans ce schéma, la liste de requêtes se vide dans celle des libérées. Il suffit donc que, lorsqu'un *thread* trouve vide la liste des requêtes libres, il permute ces deux listes[4]. Initialement, un seul verrou assurait l'exclusion mutuelle des différentes structures de gestion des requêtes. Dans le but de diminuer les conflits sur ce verrou, nous avons introduit un verrou par liste.

Le premier verrou m_1 controle l'accès à la liste de requêtes libres. Seuls les *threads* initiant une communication y sont en conflit. Le second verrou m_2 contrôle l'accès à la liste de requêtes libres. Seuls les *threads* ayant acquitté une communication y sont en conflit. Le verrou m_3 contrôle la liste de requêtes en cours.

Lorsque le démon de communication est exécuté, il parcourt la liste de « requêtes » en vérifiant ces états d'avancement. Si une requête est terminée, le démon la supprime de la liste de requêtes et le signale au *thread* propriétaire. La compétition sur ce verrou est entre les *threads* ayant initié une communication (insertion de la requête) et le démon qui la parcourit et y supprime les requêtes terminées.

Verrouillage efficace

L'amélioration visée par l'emploi de plusieurs verrous, proviendrait de la réduction de la granularité des périodes de verrouillage et des conflits sur les verrous. L'amélioration obtenue s'est révélée insuffisante. Il y avait encore un trop grand nombre de commutations de *threads* du fait de conflits sur des verrous pris. Nous

[4]Si les deux sont vides un nouveau bloc mémoire est alloué pour les requêtes.

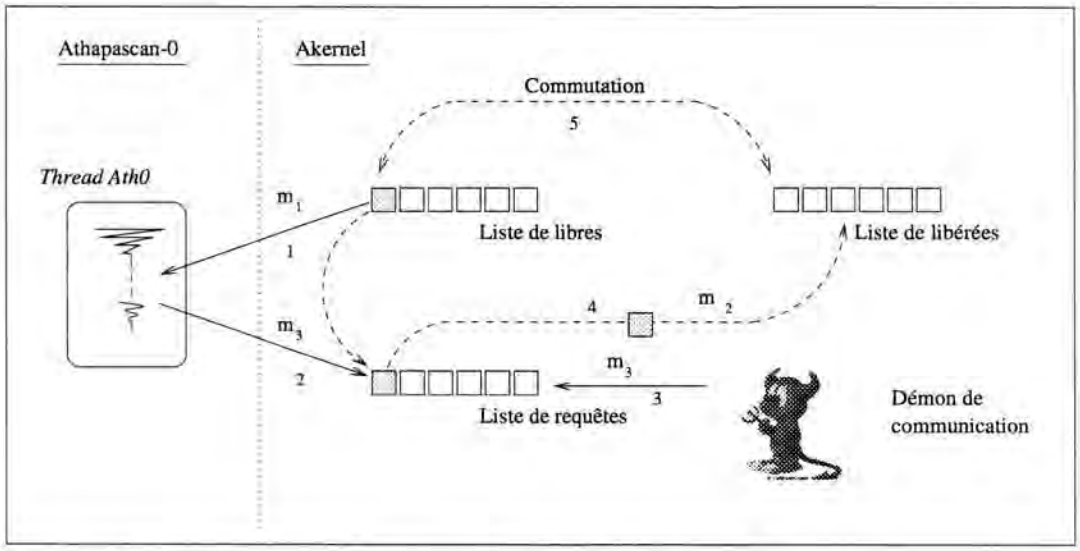

Figure 5.1 *Les structures de données de la gestion des requêtes*

avons procédé au remplacement des verrous standards POSIX par des verrous de type *spin lock* tel que ceux décrits par les programmes 5.2 et 5.4. Cette modification a amélioré de façon sensible l'efficacité de la synchronisation de la version ATHAPASCAN-SMP. Nous en présenterons les résultats au chapitre 6.

D'un point de vue pragmatique, la mise en œuvre d'ATHAPASCAN-SMP doit encore prendre en compte les conditions d'utilisation. Par exemple, sur une grappe mixte, comportant des machines monoprocesseurs et des multiprocesseurs, l'emploi de la technique de *spin lock* représente une grave source d'inefficacité sur les monoprocesseurs. Il faut donc pouvoir utiliser sur un nœud de la grappe la meilleure technique de verrouillage (*spin lock* versus *lock* standard). Ceci est décidé de façon dynamique au démarrage de l'application sur un processeur selon qu'il est monoprocesseur ou multiprocesseur.

5.2.2 Politiques de scrutation

Compte tenu des noyaux POSIX cibles, nous avons très peu de marge de réglage pour réduire l'influence de la scrutation sans pour autant nuire à sa réactivité. Le seul disponible consiste à essayer de limiter son surcoût s'il est trop important. La méthode de scrutation proposée par ATHAPASCAN-SMP présente donc quatre modes différents de déclanchement :

- par un *thread* spécialisé.
- à la demande.
- sur interruption de temps (scrutation périodique).
- sans scrutation.

Le premier mode emploie un *thread* spécialisé (démon de communication) pour exécuter la scrutation. Il est souhaitable qu'il soit réactif mais sans consommer trop de temps processeur au detriment des *threads* de calcul. Dans les SMP disponibles (Solaris, linux) le noyau de *threads* POSIX ne permet pas de contrôler l'ordonnancement. De plus l'ordonnancement réalisé n'est pas précisé au déla de l'indication que l'ordonnanceur assure un partage équitable du temps de calcul. Par conséquent, les seules garanties que nous avons sont le fait que le démon de scrutation utilise un pourcentage de calcul équivalent aux *threads* de calcul ($1/(k+1)$ pour k *threads*), et qu'il s'exécute par une période de temps borné par le *quantum*. La fréquence de scrutation est inconnue.

Intuitivement on peut affirmer que la méthode de scrutation par *thread* spécialisé provoque un problème d'efficacité lorsqu'on exécute une application dont la fréquence de communication est faible. Pour limiter les exécutions inutiles de la procédure de scrutation nous avons inclus les méthodes de scrutation à la demande et de scrutation péridiodique. La scrutation à la demande transfère la responsabilité de scruter le réseau à l'application. Dans ce cas, la réactivité et l'efficacité sont fortement dépendantes de comment l'application effectue la scrutation. La scrutation périodique considère que la scrutation est réalisé régulièrement en fonction d'un délai fixe. Cette méthode exploite le mécanisme d'interruption de temps disponible sur la plupart des systèmes. Ici, la réactivité et l'efficacité sont contraintes par la précision offerte par l'horloge système.

Aux sections suivantes nous allons décrire et analyser ces trois modes de fonctionnement. En effet le quatrième mode (sans scrutation) est un fonctionnement dégénéré d'ATHAPASCAN-SMP où il est possible de supprimer la scrutation si l'application ne fait aucune communication. C'est souvent le cas pour les applications parallèles destinées à tourner sur un seul nœud SMP. Ces modes sont choisis au moment du lancement de l'application à partir de paramètres de la ligne de commande. Le mode défaut est celui de scrutation par *thread* spécialisé.

5.3 Modèle de coût pour la scrutation par *thread* spécialisé

L'objectif de cette section est de caractériser le surcoût introduit par la méthode de scrutation par *thread* spécialisé. Pour cela nous allons considérer certaines grandeurs caractéristiques employées par ATHAPASCAN-1 [81] dans leur modèle de coût :

- T_s, la durée d'exécution séquentielle d'une application, elle ne contient aucun surcoût de parallélisme.
- T_1, la durée d'exécution du code parallèle sur un seul processeur. Cette grandeur prend en compte le surcoût introduit par la description du parallélisme : $T_1 > T_s$.

- T_p, la durée d'exécution du programme parallèle sur p processeurs. ($T_p = T_1/p$). On suppose que les calculs sont uniformément répartis sur les processeurs ;

Dans la suite, il nous faut rendre compte de la découpe d'un calcul en un nombre fixe de *threads* ainsi que de l'effet du démon de communication.

Modèle de coût : Etant donné un calcul de durée T_c à effectuer, nous pouvons écrire que le temps total (T_t) de son exécution par **une tâche** ATHAPASCAN-0 est $T_t = T_c + T_d$, où T_d est le temps de calcul de démon (*thread* spécialisé). Si l'on suppose qu'une fraction f seulement de T_c peut être réalisée en parallèle, nous pouvons réécrire T_t :

$$T_t = fT_c + T'_d + (1-f)T_c + T''_d \tag{5.1}$$

où T'_d et T''_d caractérisent l'influence du démon sur les fractions parallèle et séquentielle. En considérant qu'on décompose fT_c en k sous calculs de durée équivalente, chacun d'eux pris par un *thread*, on obtient un temps d'exécution par *thread* de fT_c/k. Or comme le temps d'exécution du démon tend à être du même ordre de grandeur que celui d'un *thread* de calcul nous pouvons supposer :

$$T'_d = f\frac{T_c}{k} \qquad et \qquad T''_d = (1-f)T_c$$

En substituant T'_d et T''_d à l'équation 5.1 nous obtenons :

$$T_t = fT_c + f\frac{T_c}{k} + (1-f)T_c + (1-f)T_c$$

$$T_t = \frac{(k+1)}{k}fT_c + 2(1-f)T_c \tag{5.2}$$

Sur un multiprocesseur le temps de calcul T_t est partagé entre p processeurs, donc :

$$T_t = \frac{(k+1)}{(kp)}fT_c + \frac{2(1-f)T_c}{p} \tag{5.3}$$

Le deuxième terme de l'équation 5.3 correspond à la fraction séquentielle de T_t donc non parallélisable pour un nombre de processeurs supérieur ou égal à 2. Pour prendre compte cet effet nous pouvons réécrire T_t comme :

$$T_t = \frac{(k+1)}{(kp)} f T_c + (1 + \lfloor \frac{1}{p} \rfloor)(1-f)T_c \tag{5.4}$$

Ce modèle simple ne prend pas en compte les délais d'attente entre les *threads* dus aux synchronisations, ni au surcoût d'ordonnancement. Il nous donne simplement une borne inférieure et supérieure pour le temps d'exécution d'une tâche ATHAPASCAN en supposant que la fraction parallélisable est toujours partagée équitablement entre les k *threads*. Il considère aussi que le nombre de *threads* est supérieur ou égal au nombre de processeurs. Dans le cas contraire ($k < p$) nous pouvons réduire l'équation 5.4 à :

$$T_t = f\frac{T_c}{k} + (1-f)T_c \tag{5.5}$$

L'accélération obtenue par une tâche ATHAPASCAN-0 par rapport au temps de calcul initial T_c peut donc être exprimée par :

$$S = \frac{T_c}{T_t} = \frac{T_c}{\frac{(k+1)}{(kp)} f T_c + (1 + \lfloor \frac{1}{p} \rfloor)(1-f)T_c} = \frac{p \times k}{k+1} \quad \text{pour} \quad f = 1 \tag{5.6}$$

Le temps d'exécution parallèle (T_p) d'une application ATHAPASCAN-0, composé par un nombre n de tâches, est donné par :

$$T_p = max(T_{t_i}) \quad i \in \quad \{\text{tâches}\} \tag{5.7}$$

De l'équation 5.4 nous pouvons déduire que l'augmentation du nombre de *threads* (k) réduit le surcoût du démon. Cette réduction est une conséquence du fait que le démon s'exécute avec une périodicité moyenne approximative de $(k+1) \times q$, où q est le *quantum*[5] pour une utilisation du SMP par la seule application.

Expérimentation : L'évaluation du surcoût introduit par le démon est relativement facile à observer. Pour l'estimer, nous avons procédé à deux tests en utilisant trois implantations distinctes du calcul de fractale de Mandelbrot (annexe C) : une version séquentielle et deux versions parallèles en employant les primitives pour la multiprogrammation légère offertes par ATHAPASCAN-SMP et par POSIX. Le premier test a consisté à comparer l'accélération obtenue par les versions parallèles par rapport à la version séquentielle. La figure 5.2 montre les résultats obtenus sur des biprocesseurs.

[5] Souvent inconnu.

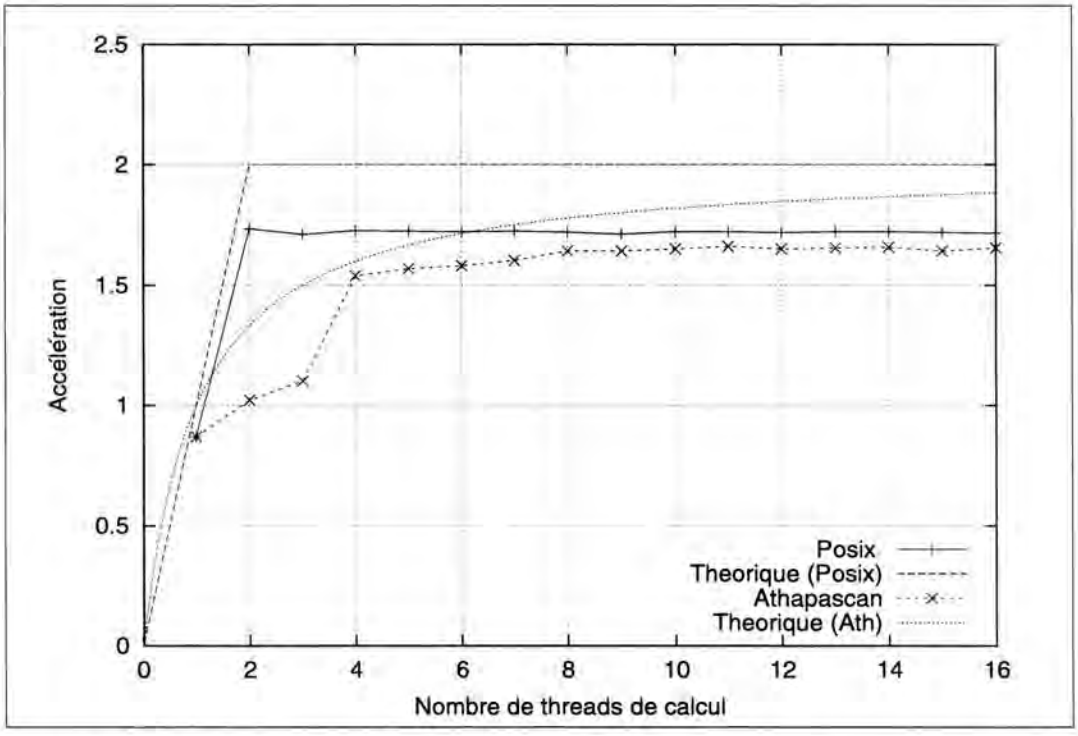

Figure 5.2 *POSIX* threads *et* ATHAPASCAN *: comparaison entre les accélerations obtenues par rapport aux valeurs théoriques (Test réalisé sur biprocesseur)*

En analysant les courbes de la figure 5.2, on observe initialement un écart important entre l'accéleration théorique et la valeur mesurée. Cet écart est en partie dû au fait que les implantations de Mandelbrot sont différentes. Les deux versions parallèles comportent une fonction de répartition des régions aux *threads* qui impliquent des synchronisations supplémentaires. En d'autres termes, on réalise une comparation entre T_1 et T_s.

Pour le cas POSIX *threads* la meilleure accélération est pour deux *threads*. A partir de ce point-là on observe une légère dégradation. Ceci provient du fait que ce test a été conduit sur une machine biprocesseur. Donc à partir de deux *threads* nous avons forcément un coût additionnel de changement de contexte dû à l'ordonnancement de plusieurs *threads* sur les deux processeurs.

Sur la courbe ATHAPASCAN-SMP, on notera une différence importante entre les valeurs théoriques (équation 5.6) et les valeurs mesurées pour un nombre petit de *threads*. Nous expliquons cette différence par le fait que notre modèle simple considère que le temps est parfaitement partagé entre les *threads*. Ce n'est pas vrai pour des *threads* à grain fin. Par exemple, si deux *threads* doivent exécuter chacun un calcul de taille T, le démon tournera aussi T unités de temps. Si cette durée est inférieure au *quantum*, un biprocesseur exécutera les 3 *threads* en $2 \times T$ ce qui est supérieur aux prédictions $(3/2 \times T)$. En augmentent la granularité de calcul, de façon à ce que les *threads* de calcul utilisent plusieurs *quanta*, nous réduisons cet

effet et provoquons un entrelacement d'exécution entre les *threads* de calcul et le démon. Ceci nous rapproche du modèle de l'équation 5.6. Nous avons pu constater ce phénomène expérimentalement.

Le deuxième test est une comparaison entre la performance obtenue par les *threads* ATHAPASCAN-SMP par rapport aux POSIX *threads* (ne pas oublier que pour la version ATHAPASCAN-SMP le démon tourne à vide). La figure 5.3 présente les résultats obtenus par les exécutions de ces deux versions sur différentes machines (monoprocesseur, biprocesseur, et un quadriprocesseur). Ces courbes donnent une performance relative c'est à dire qu'on considère que 1 est le temps de référence donné par la version POSIX (courbe « ref. »). Les autres courbes représentent le rapport entre le temps d'exécution de la version POSIX *threads* et celui de la version ATHAPASCAN-SMP.

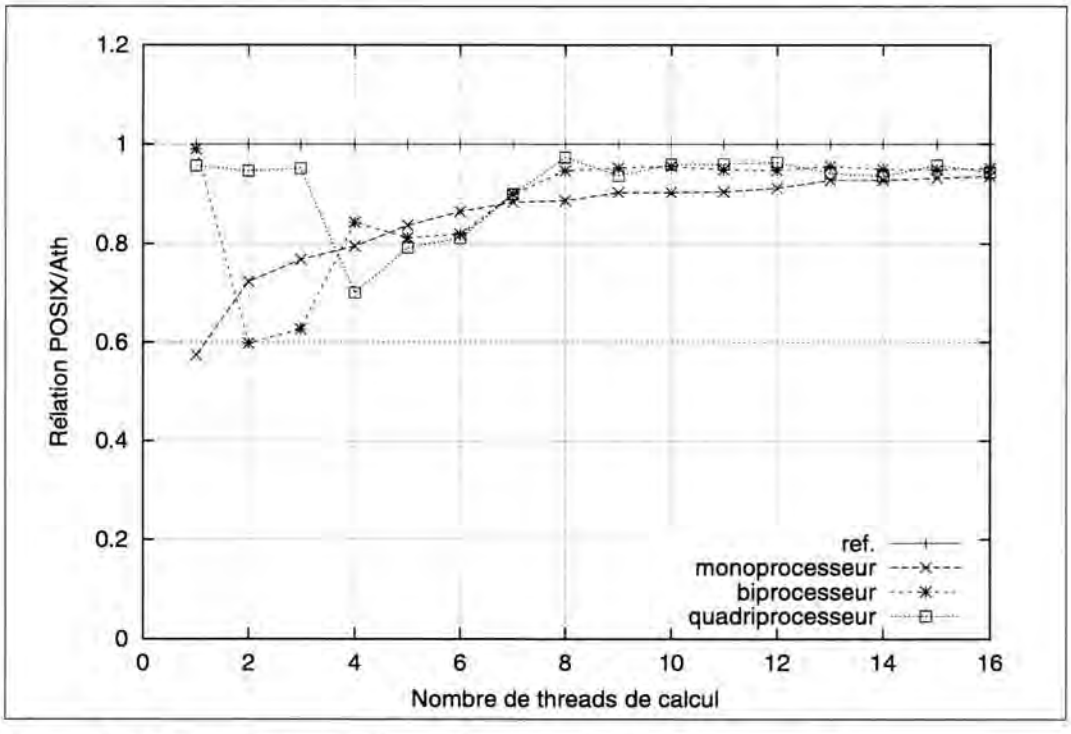

Figure 5.3 *Perturbation du démon de communication dans une application Athapascan-0 sans appels aux communication*

En analysant les courbes obtenues (fig. 5.3) nous vérifions que la version ATHAPASCAN-SMP est moins efficace que celle de POSIX *threads* sur toutes les machines. Cette différence est due au « vol » de temps d'exécution fait par le démon de communication. Nous observons aussi que le coût du démon est masqué en présence des multiprocesseurs pour un nombre de *threads* de calcul inférieur au nombre de processeurs. En effet dans ce cas, le calcul peut se faire en parallèle à la scrutation. Pour un nombre de *threads* identique au nombre de processus on vérifie une dégration de performance d'ATHAPASCAN-SMP. Cette dégradation est due au *thread* démon comme nous l'avons déjà vu auparavant (fig. 5.2).

5.4 Evaluation des politiques de scrutation d'ATHAPASCAN SMP

Le modèle de coût de la scrutation présenté précédement est très simple et ne considère pas les effets collatéraux de synchronisation imposés par les communications. Ces phénomènes sont très difficiles à modéliser. Nous allons donc observer le comportement des politiques de scrutation réseau d'ATHAPASCAN-SMP en présence d'une application réalisant des communications. La question qui se pose est alors : en présence de *threads* et de communications, ce surcoût est-il « compensable » par un gain de performance ?

Pour répondre à cette question, nous allons nous servir de l'exemple du calcul de Mandelbrot synthétique, dans sa version « par échange de message » (annexe C), comme application test pour montrer l'influence de chacune de ces politiques sur la performance. Ici un processus maître distribue du travail (régions du plan) à des processus esclaves lorsqu'ils sont inactifs. Pour un plan $N \times N$ le temps de calcul est donc estimé par $T = P + C$, où P est le temps de calcul et C le temps de communication. Nous pouvons écrire T comme :

$$T = \sum_{r=1}^{N_r} (P_r + C_r) \tag{5.8}$$

où N_r est le nombre de régions qui composent le plan, P_r le coût de calcul de la région r, et C_r le coût de communication entre le maître et l'esclave. Si l'on considère n comme la dimension d'une région r nous pouvons réécrire T_t :

$$P = \sum_{r=1}^{N_r} n^2 \times \tau \qquad \text{et} \qquad C = \sum_{r=1}^{N_r} (n^2 \times t_b + t_i) \tag{5.9}$$

où τ est le coût de calcul d'un point du plan, t_b est le temps d'envoi d'un octet ($n^2 \times t_b$ est le temps d'échange de l'esclave vers le maître) et t_i représente les données envoyées du maître vers l'esclave. La fonction T peut donc être réécrite comme :

$$T = N^2(\tau + t_b) + N_r \times t_i \tag{5.10}$$

Cette valeur idéale doit être accrue d'une valeur δ représentant les délais d'ordonnancement système et le surcoût de gestion des communications.

Notre objectif est d'évaluer l'évolution de ce surcoût dans différents régimes calcul-communication (nombre de régions). Il nous faut donc éliminer toute vari-

ation de performance pouvant être attribuée à la distribution du calcul sur des différents processeurs pour ne conserver que l'effet des communications et l'ordonnancement des *threads*. Le placement des processus est donc le suivant : le maître est placé sur un processeur et les esclaves sur l'autre processeur.

Notre expérience est l'évaluation Mandelbrot synthétique pour différentes dimensions du plan (32, 64, 128, 256 et 512). On fixe la taille de la région (32) de façon à maintenir constant le coût de calcul et le nombre de données échangées par région. Augmenter la dimension du plan se traduit par augmenter par un facteur puissance 2 le volume de calculs et de communications réalisés sans modifier leur granularité. On introduira aussi des *threads* pour calculer en parallèle les régions et on analysera l'impact de cette parallélisation en utilisant chacune des politiques de scrutation.

Pour mieux comprendre les temps impliqués il faut clarifier deux points de notre implantation de Mandelbrot. Le calcul de chaque région signifie trois échanges de messages entre le maître et l'esclave. Initialement le maître envoie un message de quatre entiers vers l'esclave. Ceci représente les coordonnées de la diagonale de la région à calculer. L'esclave, après avoir évalué la région, renvoie ces coordonnées et ensuite le message contenant les points calculés[6]. Chaque point calculé correspond à un octect. Le second point porte sur la façon dont on gère les *threads*. On crée un certain nombre fixe de *threads* indépendemment de la taille de la région et du plan. Si le nombre de *threads* est supérieur au nombre total de régions, les *threads* excédentaires resteront bloqués en attente de calcul. Ils contribuerons cependant à la durée totale de création et de terminaison.

5.4.1 Scrutation par *thread* spécialisé

Ici la scrutation est faite par un *thread* spécialisé, le démon de communication, en suivant le schéma présenté à la section 4.2.3. Nous allons seulement nous concentrer sur les résultats obtenus par cette politique. Ceux-ci sont présentés à la figure 5.4 (attention à l'échelle logarithmique). Notre référence de comparaison est une version du programme de test Mandelbrot en utilisant MPI (échange de messages entre processus « lourds »).

En analysant la figure 5.4 nous observons que :

- Dans le cas extrême où la taille de la région correspond à la taille du plan, indépendemment du nombre de *threads* créés, un seul réalise le calcul. Le temps d'exécution du programme ATHAPASCAN-SMP varie entre $\approx 2.84ms$ (1 *thread*) à $\approx 3.3ms$ (8 *threads*) contre $\approx 2.24ms$ de MPI. La différence provient de la synchronisation de fin qui demande un message additionnel du maître vers chacun des *threads* esclaves.

[6]Il est vrai qu'on pourrait éventuellement envoyer les coordonnées et les valeurs calculées dans un seul message. Cependant on serait obligé d'emballer les 4 entiers et puis les octets. Nous avons préféré garder les coûts d'envoi de messages sans introduire ceux de l'emballage et déballage.

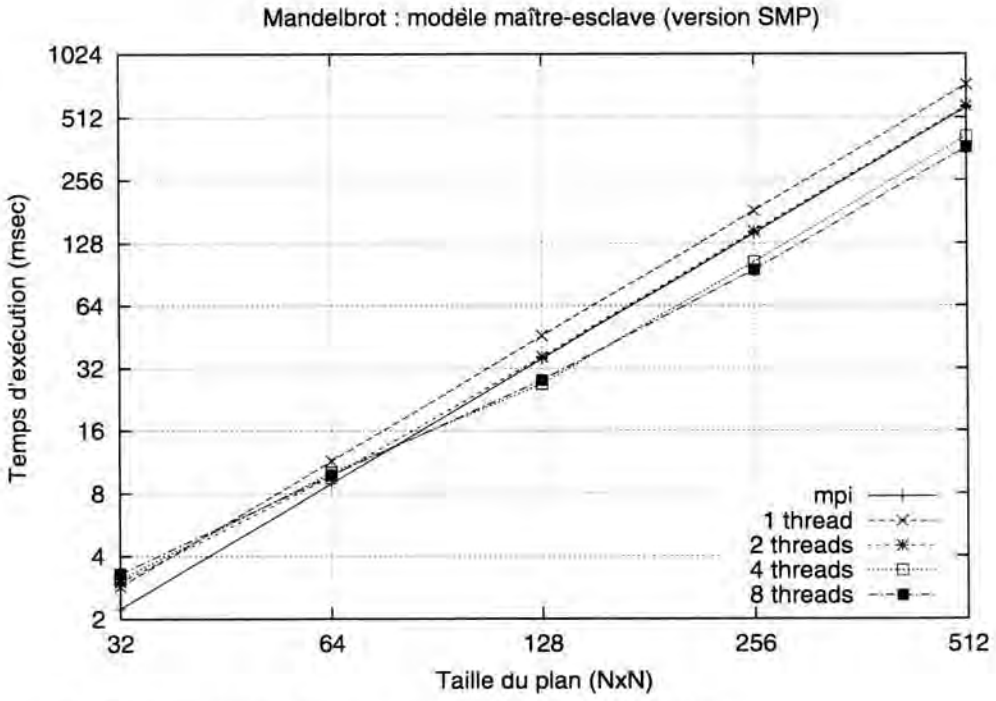

Figure 5.4 *L'évaluation de la scrutation par* thread *spécialisé (ATTENTION : échelle logarithmique)*

- L'évaluation par un seul *thread* de calcul introduit un surcoût pratiquement constant de $\approx 28\%$ par rapport à la version MPI.
- Le coût des *threads* commence à s'amortir à partir d'un plan de taille 64 soit 4 régions à calculer. Comme ce test a été conduit sur un biprocesseur, nous avons la possibilité d'avoir deux *threads* en train de s'exécuter simultanément (2 calculs ou 1 calcul et le démon de communication). Pour deux *threads* de calcul nous sommes dans la situation discutée à la section 5.2.2 qui prévoit une accéleration voisine de 1 dans le cas où la granularité de calcul est inférieure (ou de l'ordre du *quantum*). On peut d'ailleurs vérifier que la courbe relative à 2 *threads* accompagne celle de MPI.
- Pour un nombre supérieur de *threads* nous diminuons l'influence du démon (équation 5.6) pour atteindre une accéleration théorique de 1.6 pour 4 *threads* et de 1.75 pour 8 *threads*. Nous observons à partir d'un plan de taille 256, respectivement, une accéleration de 1.4 et 1.6 qui reste stable.

5.4.2 Scrutation par demande

Dans cette méthode la scrutation est réalisée par l'application. Le programmeur est responsable de l'avancement des communications. Si la scrutation est déclanchée trop souvent par rapport à la fréquence des communications, un surcoût inutile est introduit. Si elle n'est pas declanchée assez souvent, la réactivité est

insuffisante. Le programmeur peut exploiter la connaissance de l'application pour scruter au bon moment et ainsi réduire le « vol » de temps de calcul provoqué par le démon quand il est ordonnancé par le système. Le programmeur dispose de deux façons de réaliser la scrutation :

- Utiliser des appels de communications bloquants et un *thread* à part dans une boucle d'exécution, effectue la scrutation. On notera que ceci est la base de la scrutation par démon.
- Utiliser des appels de communications non bloquants et réaliser durant le calcul des appels à la fonctions de scrutation.

Cette politique nécessite la modification du programme pour qu'il puisse réaliser la scrutation. Nous avons donc modifié notre Mandelbrot pour que chaque *thread* de calcul réalise des appels non bloquants et fasse une boucle similaire à une attente active, c'est à dire une séquence de type « scruter, tester fin communications, rendre la main aux autres *threads* ». Les résultats de cette politique sont illustrés à la figure 5.5 pour la même suite de tests que pour la scrutation par *thread* spécialisé.

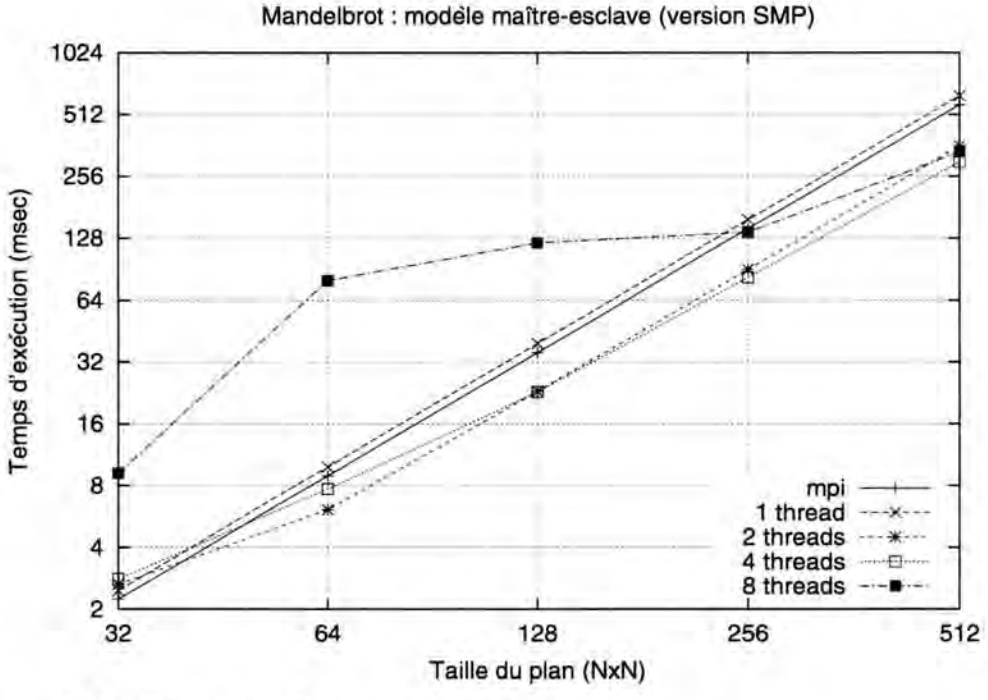

Figure 5.5 *L'évaluation de la scrutation par demande ((ATTENTION : échelle logarithmique)*

Ici nous remarquons que :

- Dans le cas où la taille de la région correspond à la taille du plan le temps d'exécution du programme ATHAPASCAN-SMP varie entre $\approx 2.47ms$ (1 *thread*) à $\approx 9.2ms$ (8 *threads*) contre $\approx 2.24ms$ de MPI. On observe que

pour 1 *thread* le coût est inférieur au cas précédent : le démon ne vole plus de temps de calcul. Par contre on remarquera l'effet de l'attente active dès que le nombre de *threads* augmente sans que le besoin de calcul suive.

- Le surcoût introduit par un seul *thread* est toujours constant mais de $\approx 8\%$ ce qui montre le temps « épargné » par l'absence de démon de communication.
- Pour deux *threads* nous obtenons une accélération de ≈ 1.6 par rapport à MPI à la place d'une accéleration proche de 1 du fait de la disparition du démon.
- La mauvaise performance pour 8 *threads* s'explique par le fait que tous les *threads* sont en train de boucler en attente d'une région à calculer. Ayant moins de régions à calculer la présence de ces *threads* est plus nuisante. Nous pouvons les voir comme n démons de communication qui « volent » du temps de calcul inutilement.
- Il est de même pour 4 *threads* jusqu'au moment où ils deviennent utiles pour une découpe d'un plan de taille 64 et plus. On remarquera que 2 *threads* sont plus performantes pour cette situation.
- L'augmentation du nombre de *threads* de 4 à 8 pour des plans supérieurs à 256 introduit une amélioration de ($\approx 8\%$) de la performance malgré le fait que le noyau doit les ordonnancer sur deux processeurs. Nous avons observé que cette écart se mantient pour un plan 1024x1024. Ayant plusieurs *threads*, il en aura toujours un prêt à s'exécuter.

En comparant les résultats obtenus par les deux politiques de scrutation (par *thread* spécialisé et à la demande) la scrutation à la demande est plus performante pour un petit nombre de *threads* de calcul. Nous avons mesuré un gain d'environ 40% pour 2 *threads* de calcul et seulement 10% pour le cas de 8 *threads* de calcul. Cette différence de gain est expliquée par le fait que l'augmentation du nombre de *threads* réduit l'influence de « vol » de temps de calcul par le démon. La grand désavantage de la scrutation par demande est le fait que le programmeur doit s'occuper de la scrutation. Une mauvaise politique d'implantation provoque de graves problèmes de performance. Le programmeur doit encore considèrer la granularité par rapport au nombre de *threads* et le besoin de communication. Par exemple, sur le cas analysé, la présence de 8 *threads* s'est avérée « payante » seulement à partir d'une dimension de plan supérieure à 256. Tous ces détails sont une complication additionnelle à la programmation.

5.4.3 Scrutation périodique

S'il n'est pas possible d'augmenter le nombre de *threads* sans accroître le nombre de communications et diminuer la taille de messages, on peut essayer de limiter l'activité du démon en retardant son exécution d'un délai fixe. C'est la scrutation périodique. La fixation du délai est contrainte par la précision de l'horloge du système utilisé. Le système Solaris 2.6, par exemple, fixe une valeur minimum de $20ms$

et un incrément de $10ms$ (fig. 5.6). Par conséquent la période minimum d'activation contrôlée du démon sera donc toujours supérieure à $20ms$.

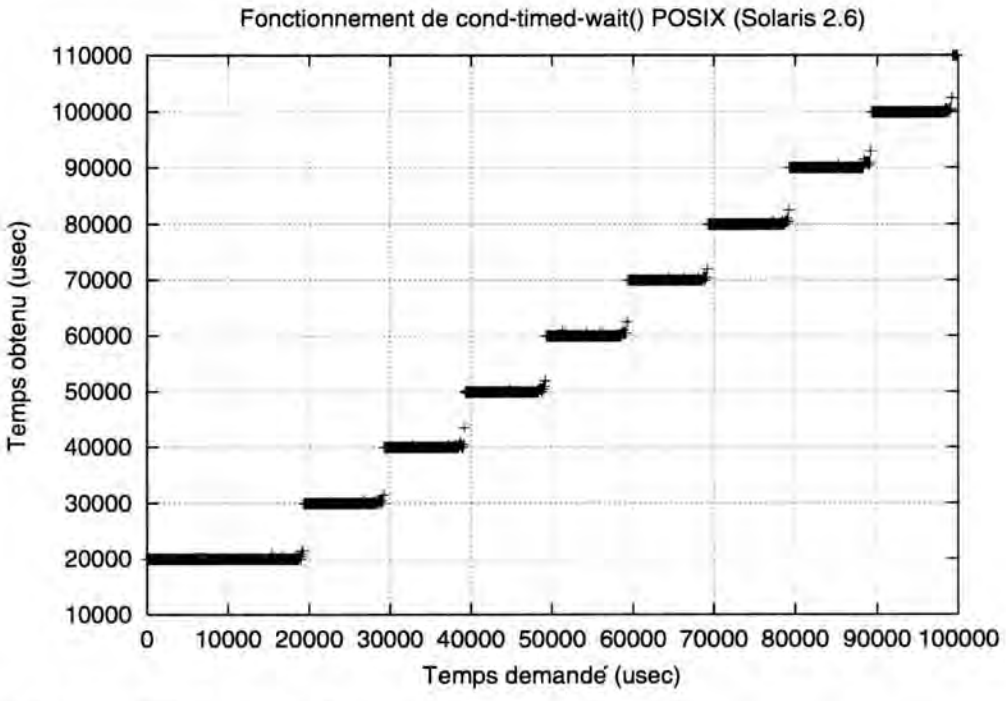

Figure 5.6 *L'évaluation de la scrutation périodique*

L'introduction de ce dispositif de réduction de l'activité du démon a mis en évidence un problème dans la scrutation des requêtes faite par le démon. Dans l'implantation d'ATHAPASCAN (les deux versions) le démon s'arrête à la première requête prête qui est rencontrée. Dans ce mode de scrutation, il ne traite une requête que toutes les $20ms$ au mieux. Pour le test de Mandelbrot tel que nous l'avons implanté, ceci implique au moins $60ms$ pour chaque région évaluée (3 échanges de messages). Il faut donc, au moins, prévoir un consultation de toute la liste des requêtes pour essayer d'avancer le plus possible de *threads* en attente de communication avant que le démon s'endorme à nouveau[7].

5.5 Bilan

Nous avons proposé une implantation d'ATHAPASCAN-SMP à destination des machines SMP. La performance insuffisante de la version initiale d'ATHAPASCAN-SMP venait principalement d'un problème classique : la granularité de verrouillage et par conséquent des conflits d'accès aux verrous. Nous avons donc proposé une nouvelle façon de gérer le verrouillage par l'introduction de plusieurs verrous et par l'emploi de la technique de *spin lock*. A première vue cette nouvelle gestion

[7]Ceci n'est pas implanté à l'heure de rédaction de ce document...

des requêtes peut apparaître plus coûteuse car elle oblige plusieurs opérateurs de verrouillage au lieu d'une seule dans l'implantation initiale. Ainsi l'initiation d'une communication par un *thread* nécessite plusieurs verrouillages disjoints consécutifs :

- verrouillage de la liste de requêtes libres.
- verrouillage de MPI.
- verrouillage de la liste de requêtes en cours.

L'expérience a montré que le taux de conflit a été diminué et que l'efficacité atteinte est supérieure. On notera cependant que les modifications introduites pour la version SMP sont au mieux inefficace sur un monoprocesseur et au pire ne fonctionnent pas sur un monoprocesseur, ce qui nécessite deux versions d'ATHAPASCAN-0.

S'il a été possible de contrôler la dégradation de performances due aux conflits d'accès sur les données partagées il n'a pas été possible de fournir un mécanisme de réglage du processus de scrutation en charge de l'avancement des communications. La cause essentiellement est l'absence d'indicateur précis sur les changements d'état des dispositifs de communication et le manque de contrôle précis de l'ordonnancement des *threads*. Ceci se traduit par :

- L'impossibilité de gérer les priorités à partir du niveau applicatif. Dans la plupart des noyaux POSIX utilisant un modèle 1 :1, le noyau de *threads* gère dynamiquement les priorités de *threads* de façon à faire un partage équitable du temps de calcul. Un *thread*, parmi k, recevra toujours (à long term) $1/k$ du temps processeur.
- L'emploi de l'appel *yield()* ne permet pas à un *thread* de passer le contrôle à un autre *thread*. Ce passage n'a lieu en effet que si le *thread* s'exécute « un certain temps » dépendant du système.
- La résolution du mécanisme d'horloge offerte par la plupart des systèmes n'est pas assez fine. Par conséquent, bloquer la scrutation par une durée de temps fixe (via *cond_timed_wait()* POSIX, par exemple) peut amener à des temps beaucoup plus importants que ceux souhaités.

Ces problèmes sont à l'origine du surcoût du démon de communication même en l'absence de communication. Les expériences préliminaires ont montré la faisabilité de la scrutation « à la demande » ou « par démon » Le mode par défaut d'ATHAPASCAN-SMP est celui de la « scrutation par démon ». Nous croyons que le surcoût du démon ($1/(k+1)$ en présence de k threads) est compensé par une programmation plus simple.

Déterminer l'importance de ce surcoût et la réactivité d'une telle méthode de scrutation font partie de l'évaluation globale des performances d'une grappe de SMP. Ceci est fait au chapitre suivant.

6

Évaluation de performance

"Il ne faut jamais vendre la peau de l'ours avant qu'on ne l'ait mis par terre"
Jean de La Fontaine, L'ours et les deux compagnons

Dans ce chapitre nous allons évaluer ATHAPASCAN-SMP pour montrer que les modifications que nous avons apportées à la version originale se révèlent efficaces sur un multiprocesseur. Nous étudierons initialement le comportement d'ATHAPASCAN-0 et d'ATHAPASCAN-SMP vis à vis des noyaux de base, POSIX *threads* et MPI, afin de mettre en évidence les surcoûts introduits. Ensuite nous comparerons la version originale d'ATHAPASCAN-0 et la version SMP. Nous partirons des critères définis par I. Ginzburg [86] pour l'évaluation initiale d'ATHAPASCAN-0 sur le SP1. Ensuite nous analyserons plus précisément les méthodes de scrutation par *thread* spécialisé et à la demande pour différentes découpes calcul-communications. Nous terminerons en montrant le gain de performance apporté par ATHAPASCAN-SMP sur certaines applications réelles.

6.1 Méthodologie d'évaluation

Un des objectifs de l'évaluation de performance est d'analyser le comportement d'un système par rapport à certaines caractéristiques, et éventuellement de le comparer avec d'autres systèmes. Cette évaluation est basée sur l'analyse de séries de mesures expérimentales faites sur le système cible. Pour une caractéristique, aucune conclusion correcte ne peut être faite de façon absolue à partir d'une seule série des mesures du fait des fluctuations de celles-ci [101]. Il faut donc faire un traitement statistique de ces données afin de déterminer pour chaque valeur caractéristique son

domaine de fluctuation. La technique utilisée tout au long de ce document, et en particulier dans ce chapitre, est celle du calcul d'un **intervalle de confiance**. Cette technique est brièvement présentée à l'annexe A. Toutes les données présentées par la suite le sont avec un niveau de confiance de 90%.

6.2 ATHAPASCAN-0 versus ATHAPASCAN-0 SMP

Cette section est consacrée à l'évaluation d'ATHAPASCAN-SMP sur des multiprocesseurs. Il est confronté à l'implantation originale d'ATHAPASCAN-0 et aux opérateurs équivalents d'un noyau 1 :1 POSIX *threads* et de la bibliothèque MPI si possible. Cette confrontation est destinée à vérifier le gain de performance obtenu par les adaptations spécifiques aux SMP.

Les tests detaillés par la suite sont ceux effectués sur multiprocesseurs. En effet, les tests effectués sur monoprocesseurs (le *spin lock* est remplacé par un verrouillage classique) ont montré que les deux implantations étaient équivalentes[1].

La méthode de scrutation utilisée lors des tests est celle utilisant un démon spécialisé. En effet, les problèmes d'efficacité et de réactivité d'ATHAPASCAN-0 sur multiprocesseurs ont leur origine dans le fait que le démon de communication rentre en compétition avec les *threads* de calcul. La méthode de scrutation à la demande est faite au niveau applicatif et élimine ce problème. Les deux versions d'ATHAPASCAN sont équivalentes[2] dès que la scrutation est faite au niveau applicatif. Enfin, nous avons écarté l'utilisation de la scrutation périodique. Elle dépend de la résolution de l'horloge du système qui sur notre plate-forme est supérieure à $20ms$ et induit une fréquence de scrutation trop faible.

L'évaluation d'ATHAPASCAN-SMP utilise des tests basés sur ceux effectués par I. Ginzburg dans son travail de thèse [86] décrivant l'évaluation de la version initiale de ATHAPASCAN-0 sur la machine IBM SP1. La démarche adoptée consiste à mesurer les temps d'exécution des primitives ATHAPASCAN-0 dans un environnement non chargé, c'est à dire celle où la primitive à l'étude s'exécute en dehors des *threads* nécessaires au fonctionnement d'ATHAPASCAN-0. Ces tests sont divisés en deux groupes, celui des fonctions de base et celui des fonctions composées, que nous présentons à la suite.

[1] Le critère de comparaison est celui fourni à l'annexe A : deux systèmes sont équivalents si la moyenne d'une valeur caractéristique d'un système est comprise dans l'intervalle de confiance de l'autre. Ceci est le cas pour les deux version d'ATHAPASCAN (intervalle de confiance à $\alpha = 90\%$). On conclut donc que le surcoût introduit par la version SMP dû aux verrouillages disjoints et consécutifs de la nouvelle gestion de requêtes n'est pas significatif.

[2] Selon le même critère d'intervalle de confiance.

6.2.1 Fonctions de base

Notre objectif est donc de mesurer les surcoûts introduits par ATHAPASCAN-0/SMP par rapport aux noyaux de base. Dans un premier temps nous évaluerons les fonctionnalités pour la multiprogrammation légère. Le point clef d'ATHAPASCAN est d'être une bibliothèque de communications *thread aware*. Pour y arriver, nous avons encapsulé les primitives MPI (cf. section 4.4.2). Ce mécanisme implique un surcoût de gestion que nous voulons estimer. Nous analyserons aussi le coût d'exécution des créations de *thread* à distance qui sont des apports d'ATHAPASCAN.

La multiprogrammation légère ATHAPASCAN-0

ATHAPASCAN-0 offre au niveau de l'interface de multiprogrammation légère un ensemble de primitives très semblable à celles du standard POSIX *threads*. Les *threads* ATHAPASCAN-0 sont construits au-dessus des *threads* du système cible (section 4.4.2, «La réalisation d'ATHAPASCAN-0»). On peut espérer donc une performance similaire à celles de la bibliothèque de base. Les seules exceptions sont pour les primitives de création de *thread* (local ou à distance) et pour les sémaphores.

Le tableau 6.1 présente les valeurs mesurées pour l'utilisation d'un *thread* ATHAPASCAN-0 local (esclave), pour la commutation de contexte, et pour les primitives de synchronisation. Les valeurs présentées sont significatives avec un intervalle de confiance de $\alpha = 90\%$. Les tests effectués sont ceux présentés au chapitre 2 (section 2.6.1).

Tous les opérateurs ATHAPASCAN-0[3] montrent un surcoût. La création d'un *thread* local (esclave) implique un appel à la bibliothèque de *threads* utilisée et à l'association à ce *thread* d'informations de gestion propres à ATHAPASCAN-0, d'où son surcoût. La création des *threads* à distance sera analysée plus tard dans cette section. Pour les primitives de synchronisation, le surcoût s'explique par le fait qu'elles sont implantées par des fonctions qui comptabilisent le nombre de *threads* bloqués (statistiques internes d'ATHAPASCAN-0) et effectuent un contrôle d'erreur en plus de la primitive originale POSIX.

Les sémaphores ATHAPASCAN-0 sont implantés en utilisant les opérations de synchronisation plus primitives : *lock*, *unlock* et variables de condition. Apparemment ce mécanisme, lorsque le blocage n'intervient pas, se montre plus efficace que les sémaphores natifs[4] (POSIX 1003.1b). Cela explique aussi la différence obtenu pour le temps de commutation de contexte car nous employons des sémaphores pour forcer celle-ci (cf. section 2.6.1).

Les modifications effectuées à la version originale d'ATHAPASCAN-0 ont porté

[3] Sauf les sémaphores.

[4] Les sémaphores n'appartiennent pas à la définition de la norme POSIX *threads*.

sur la communication. Le comportement des deux versions d'ATHAPASCAN sont identiques en ce qui concerne les primitives de multiprogrammation légère (exception faite pour la création de *threads* à distance).

Tableau 6.1 *Comparaison* threads ATHAPASCAN-*0 et POSIX* threads
(Tests réalisés sur un biprocesseur Intel 333Mhz, solaris 2.6)

	ATHAPASCAN-0	POSIX *threads* (1 :1)
Création thread	195μs	156μs
Commutation de contexte	12.2μs	12.5μs
lock/unlock	0.398μs	0.325μs
trylock/unlock	0.392μs	0.325μs
trylock(verrou non disponible)	0.210μs	0.160μs
P&V(sémaphore disponible)	0.920μs	1.450μs

Les communications point à point

Nous avons évalué les indicateurs caractéristiques de communication (débit, temps d'amorçage et taille de message à mi-débit) pour les deux versions d'ATHAPASCAN en utilisant le *benchmark* COMMS1 (*ping-pong*) comme nous l'avons fait pour MPI à la section 3.6.1. Le programme de test ATHAPASCAN-0/SMP crée un *thread* pour l'envoi et la réception de messages (sur le noeud *ping*) et un autre *thread* pour la réception et le renvoi de messages (sur noeud *pong*). Les primitives utilisées sont les primitives synchrones offertes par ATHAPASCAN-0.

Ainsi comme nous l'avons fait au chapitre 3, nous avons pris en compte les plages de fonctionnement des deux protocoles d'envoi de messages, *short* et *long* offerts par LAM. Le tableau 6.2 compare les indicateurs de performance d'ATHAPASCAN-0/SMP à ceux de MPI pour ces deux situations.

A partir des figures 6.1,6.2 et 6.3 nous observons que les courbes des deux versions d'ATHAPASCAN-0/SMP sont similaires à celle de LAM MPI. Par contre, les deux versions sont moins efficaces que MPI sauf pour des gros messages (fig. 6.3). Le surcoût se manifeste surtout sur la latence d'amorçage du fait de la prise en charge des communications par le démon. Il se traduit par une augmentation de la taille des messages nécessaire pour atteindre le mi-débit.

Il est important de noter que dans le cas de figure où aucun *thread* ne calcule, la réactivité des deux versions d'ATHAPASCAN est la meilleure : le démon de communication tourne de façon pratiquement continue sur un des processeurs de la machine. La différence entre les deux versions provient des synchronisations de gestion des réveils des *threads ping* ou *pong*. D'autres points à remarquer sont :

Tableau 6.2 *Les indicateurs r_∞,t_o ,$n_{\frac{1}{2}}$ pour* ATHAPASCAN-*0,* ATHAPASCAN-*SMP et LAM MPI version 6.3*
(Biprocesseur, réseau Ethernet 100 Mpbs, mode c2c)

Protocole *short*	Débit(r_∞^{64k})	Amorçage(t_o)	Mi-débit($n_{\frac{1}{2}}$)
LAM MPI 6.3	8.2 Mo/s	195μs($\pm$0.5)	$\approx$3.0 Ko
ATHAPASCAN-0	7.8 Mo/s	495μs($\pm$6.0)	$\approx$ 4.75 Ko
ATHAPASCAN-SMP	7.9 Mo/s	335μs($\pm$1.8)	$\approx$ 4.0 Ko

Protocole *long*	Débit(r_∞^{4Mo})	Amorçage(t_o)	Mi-débit($n_{\frac{1}{2}}$)
LAM MPI 6.3	5.9 Mo/s	195μs($\pm$0.5)	$\approx$1.75 Ko
ATHAPASCAN-0	6.2 Mo/s	495μs($\pm$6.0)	$\approx$ 3.5 Ko
ATHAPASCAN-SMP	6.2 Mo/s	335μs($\pm$1.8)	$\approx$ 3.0 Ko

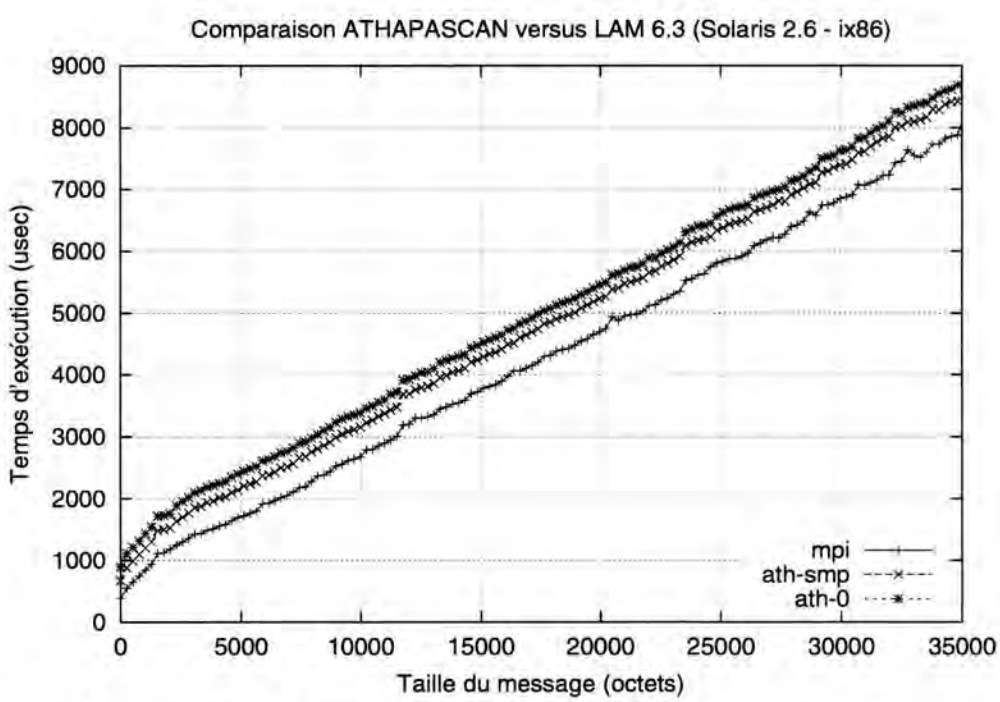

Figure 6.1 *Délai bout en bout (Biprocesseur, réseau Ethernet 100 Mpbs)*

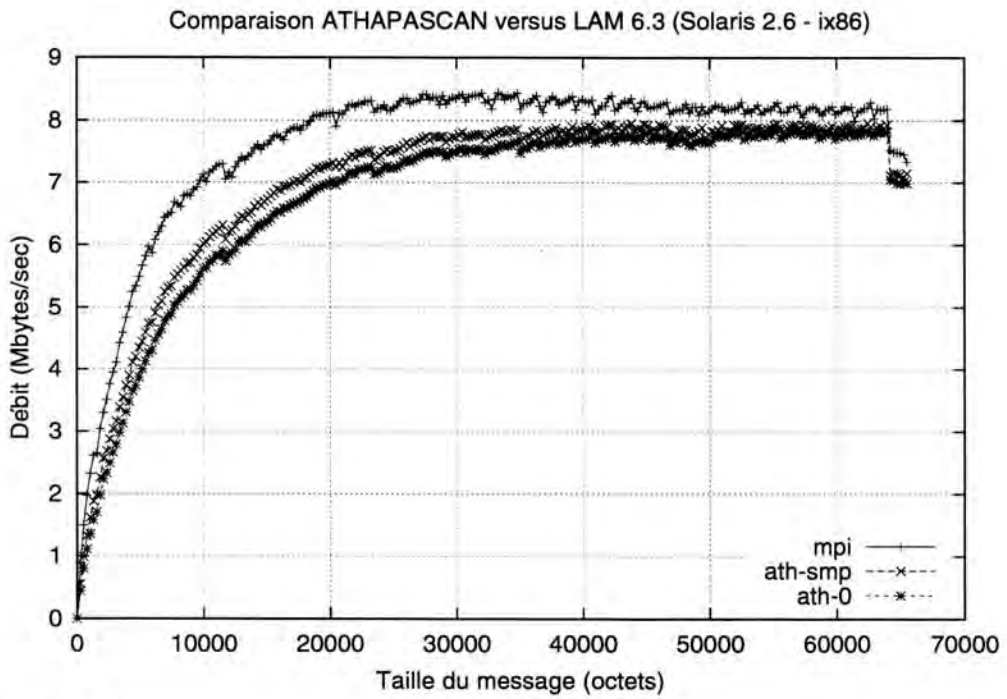

Figure 6.2 *Débit pour messages jusqu'à 64k octets (Biproc., réseau Ethernet 100 Mpbs)*

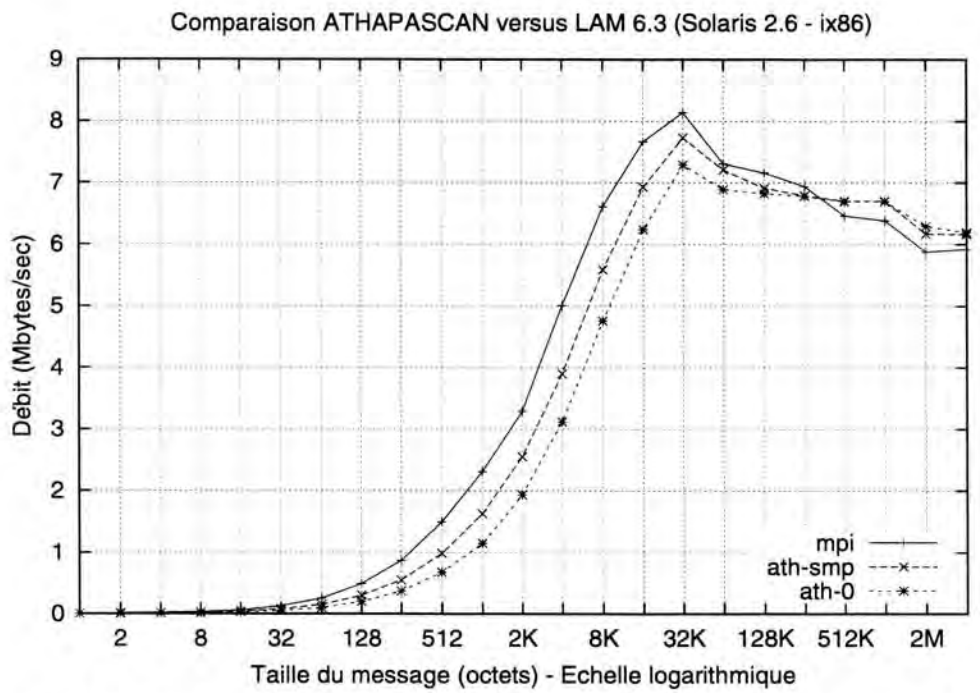

Figure 6.3 *Débit pour des gros messages (Biprocesseur, réseau Ethernet 100 Mpbs)*

- la différence initiale de pente (fig. 6.1) des courbes est due à la taille d'un paquet TCP/IP (MTU=1500 octets). Le début correspond donc à l'envoi d'un message sans besoin de fragmentation.
- la version ATHAPASCAN-SMP est plus efficace que la version ATHAPASCAN-0. Le surcoût décroît lentement avec l'augmentation de la taille des messages car l'augmentation du temps de transmission amortit le coût initial d'amorçage. Il en est de même pour la courbe de MPI.
- le gain d'environ $160\mu s$ sur le coût d'amorçage des communications constaté pour ATHAPASCAN-SMP par rapport à ATHAPASCAN-0 (fig. 6.1) peut être attribué à l'amélioration de la synchronisation due à l'introduction de verrous différenciés et l'usage de *spin lock*.
- La performance supérieure d'ATHAPASCAN-0/SMP par rapport à MPI pour de gros messages (fig. 6.3) s'explique par le fait que le test de *ping-pong* MPI utilise des primitives de communication synchrones. Or à partir de 64k octets, MPI emploie le protocole *long* c'est à dire que les messages sont envoyés après que les processus récepteurs aient posté le *receive*, ce qui provoque un blocage des processus « communicants ». Ils sont débloqués par la couche protocole système (TCP/IP) quand le message est prêt à être délivré au récepteur. Par contre, le *ping-pong* ATHAPASCAN-0/SMP est implanté par des primitives asynchrones, et le *thread* spécialisé (démon) scrute le réseau de façon continue. Ces consultations périodiques de MPI et du système éliminent les surcoûts de blocage des processus communicants. Ce phénomène est similaire à celui rapporté[5] par les développeurs de LAM lorsqu'on utilise le mode *lamd* (démon) à la place du mode *c2c* pour des gros messages.

Les appels de service

La mesure du temps d'appel des services est faite en utilisant une procédure assez similaire à celle du *benchmark* COMMS1 (*ping-pong*). Cette méthode, introduite par I. Ginzburg [86], réalise à partir de deux noeuds une séquence d'appels de services de façon mutuelle. Le noeud « ping » envoie un appel de service au noeud « pong ». Le noeud « pong » à la réception de l'appel réalise le même appel vers le noeud « ping ». Cette séquence est refaite en variant le nombre de paramètres à l'appel. A nouveau, les mesures sont faites sur des noeuds « à vide » c'est à dire sans charge de calcul. Les *threads* ATHAPASCAN ne font que communiquer entre eux par appels de services. Les résultats sont présentés aux tableaux 6.3 et 6.4, et aux figures 6.4 et 6.5. D'une manière générale, ATHAPASCAN-SMP est plus efficace que ATHAPASCAN-0. On vérifie encore que :

- la version ATHAPASCAN-SMP présente un surcoût inférieur à celui de la version ATHAPASCAN-0. Nous attribuons cette amélioration au nouveau mécanisme de gestion de requêtes et à l'emploi de l'attente active (*spin lock*).

[5] http ://www.mpi.nd.edu/lam/6.3

Tableau 6.3 *Exécution de services : les indicateurs r_∞,t_o ,$n_{\frac{1}{2}}$ - protocole* short *(Biprocesseur, réseau Ethernet 100 Mpbs, LAM version 6.3, mode c2c)*

Débit(r_∞^{64k})	Urgent	Service	Ping-pong
ATHAPASCAN-0	7.55 Mo/s	6.8 Mo/s	7.8 Mo/s
ATHAPASCAN-SMP	7.75 Mo/s	7.5 Mo/s	7.9 Mo/s

Mi-débit($n_{\frac{1}{2}}$)	Urgent	Service	Ping-pong
ATHAPASCAN-0	≈6.5 Ko	≈9.75 Ko	≈4.75 Ko
ATHAPASCAN-SMP	≈6.5 Ko	≈9 Ko	≈4 Ko

Amorçage(t_o)	Urgent	Service	Ping-pong
ATHAPASCAN-0	803μs(±5)	1932μs(±71)	495μs(±6)
ATHAPASCAN-SMP	720μs(±5)	1300μs(±5.5)	335μs(±1.8)

Tableau 6.4 *Exécution de services : les indicateurs r_∞,t_o ,$n_{\frac{1}{2}}$ - protocole* long *(Biprocesseur, réseau Ethernet 100 Mpbs, LAM version 6.3, mode c2c)*

Débit(r_∞^{4Mo})	Urgent	Service	Ping-pong
ATHAPASCAN-0	6.15 Mo/s	6.15 Mo/s	6.2 Mo/s
ATHAPASCAN-SMP	6.15 Mo/s	6.15 Mo/s	6.2 Mo/s

Mi-débit($n_{\frac{1}{2}}$)	Urgent	Service	Ping-pong
ATHAPASCAN-0	≈4.75 Ko	≈8.5 Ko	≈3.5 Ko
ATHAPASCAN-SMP	≈4.5 Ko	≈6.75 Ko	≈3 Ko

Amorçage(t_o)	Urgent	Service	Ping-pong
ATHAPASCAN-0	803μs(±5)	1932μs(±71)	495μs(±6)
ATHAPASCAN-SMP	720μs(±5)	1300μs(±5.5)	335μs(±1.8)

- le coût d'un *ping-pong* par appel service (urgent ou non) est plus élevé que celui d'un *ping-pong* « classique ». Cette différence provient de deux facteurs. Le premier est que l'exécution des services demande un emballage et déballage de données sur une structure spéciale (les tampons) - même dans le cas d'envoi de zéro paramètre - cet emballage est absent de l'échange normal de données. Le deuxième facteur, le plus significatif, est dû à la prise en charge des communications par des démons spécialisés, lesquels doivent recevoir la demande de service, la décoder, et appeler la procédure associée (service urgent), ou créer un *thread* pour la réaliser (service).
- pour le surcoût d'amorçage, nous avons observé un rapport approximatif de 2 entre l'exécution d'un *ping-pong* par service urgent et d'un *ping-pong* classique, et d'environ 4 entre un *ping-pong* service et le *ping-pong* classique. Ces valeurs sont compatibles avec les résultats obtenus par I. Ginzburg sur

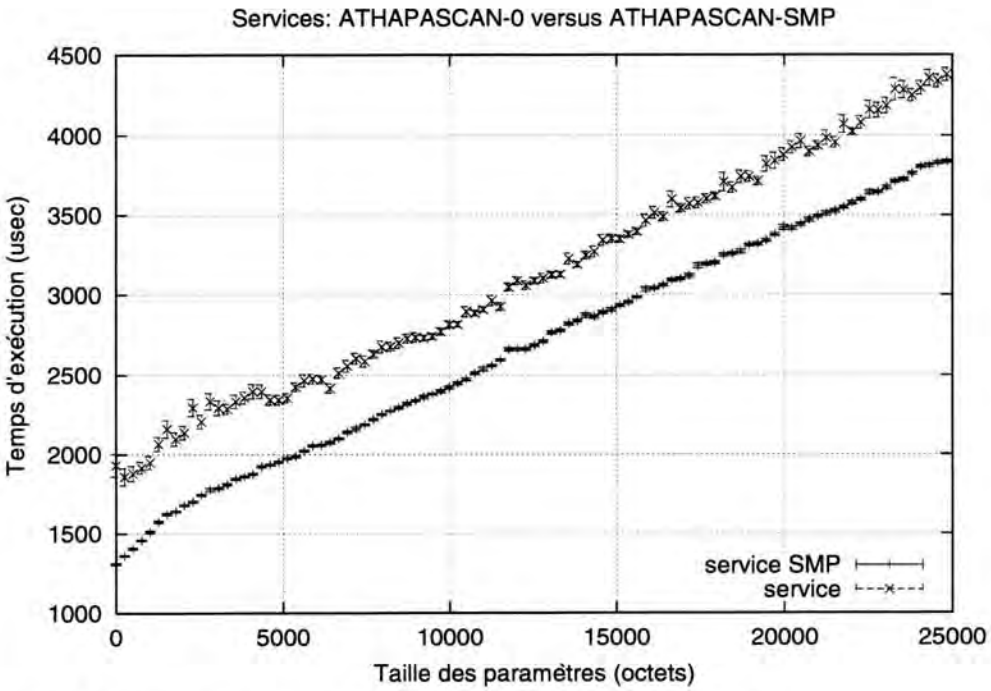

Figure 6.4 *Temps de transmission : petits messages (protocole* short*)*
(Biprocesseur, réseau Ethernet 100 Mpbs, mode -c2c)

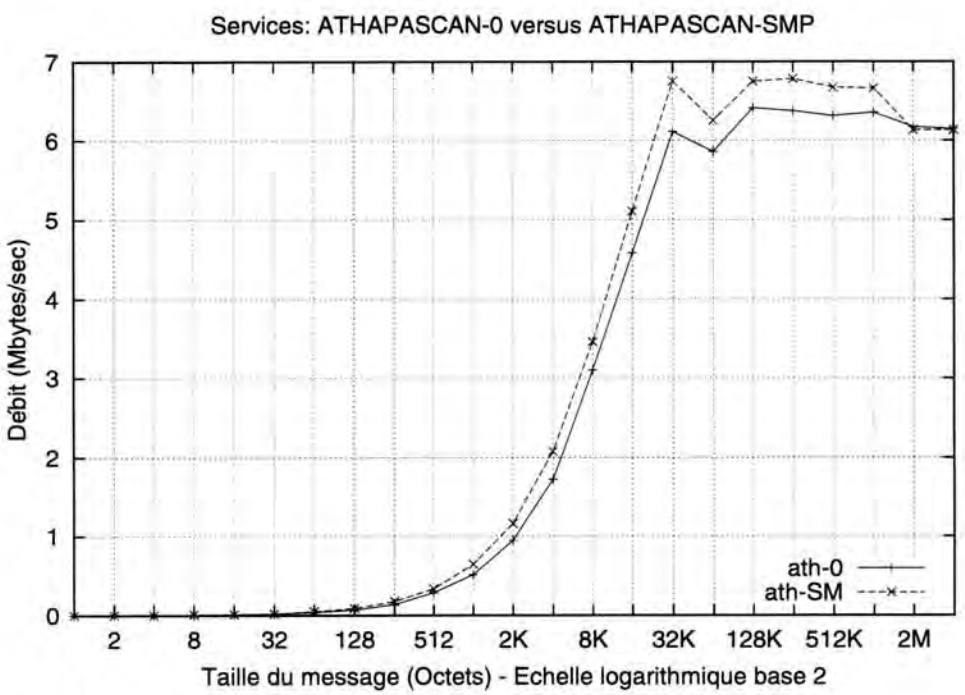

Figure 6.5 *Débit gros messages (protocole* long*)*
(Biprocesseur, réseau Ethernet 100 Mpbs, mode -c2c)

la machine IBM-SP1 et s'expliquent. Dans un *ping-pong* classique, le démon de communication ATHAPASCAN-0/SMP réveille un *thread* utilisateur en attente de communication. Pour le *ping-pong* par services urgents, le démon de communication ATHAPASCAN-0/SMP réveille le démon de services urgents lequel fera la communication. Le surcoût observé (100%) correspond au surcoût de traitement des tampons et décodage de service. Le coût de création de *thread* s'ajoute à ces coûts pour un *ping-pong* utilisant l'appel de service (surcoût d'amorçage fois 4).

- le temps d'exécution obtenu par la réalisation de services non urgents (création de *thread* à distance) par la version ATHAPASCAN-0 sont moins stables que ceux de la version ATHAPASCAN-SMP (fig. 6.4). Nous n'avons pas pu avoir deux exécutions consécutives similaires pour ATHAPASCAN-0. Ce comportement chaotique et non reproductible, nous parait relever de phénomènes d'ordonnancement des *threads*.
- Pour de gros messages ATHAPASCAN-SMP continue à être plus performant qu'ATHAPASCAN-0. Nous expliquons cela par le problème de contention de verrous. Dans ATHAPASCAN-0 le *thread* qui réalise le service rentre en compétition par le verrou utilisé par le démon. Ce verrou est aussi utilisé pour protéger l'accès à MPI, ou l'emballage des paramètres nécessaire aux services. Cet forte compétition entraîne des commutations de contexte inutiles, ce qui est éliminé en ATHAPASCAN-SMP.

Bilan partiel

Nous avons étudié les surcoûts des opérateurs ATHAPASCAN-0/SMP sur les opérateurs équivalents des noyaux de base POSIX *threads* et MPI. Les tests réalisés ont été menés sur des biprocesseurs en considérant des noeuds non chargés. Cette situation est idéale pour les deux versions d'ATHAPASCAN-0 car le *thread* spécialisé (démon de communication) tend à tourner de façon presque continue sur un des processeurs, ce qui accentue la compétition sur les verrous.

Les tests effectués ont montré que la version ATHAPASCAN-SMP est plus performante que la version originale sur des machines multiprocesseurs. Cette amélioration est due essentiellement à la diminution de la contention et de la compétition sur les verrous, et de la réduction de commutations de contexte inutiles (*spin lock*). Nous avons pu vérifier ce dernier point en observant les statistiques systèmes[6] résultants de l'exécution de ces programmes de tests.

[6]A partir du système de fichiers /proc (Solaris 2.6).

6.2.2 Fonctions composées : Intégration de communications et multiprogrammation légère

La motivation principale de l'intégration de la communication et de la multiprogrammation légère est de pouvoir facilement recouvrir calcul et communication, et même de réaliser simultanément ces actions en profitant du parallelisme réel offert par un multiprocesseur. Les tests suivants ont pour objectif de déterminer dans quelle mesure cet espoir est vérifié.

Le recouvrement de communications-communications

Le premier test est destiné à vérifier si la multiprogrammation permet d'exploiter efficacement, et simplement, les possibilités de recouvrement offertes par la bibliothèque de communication (cf. section 3.5.4). On essaie donc de vérifier que le délai d'attente de fin de communication d'un *thread* peut servir à faire avancer une autre communication. Le test effectué est une variante de COMMS1 où, sur deux noeuds, plusieurs paires de *threads* effectuent un *ping-pong*[7]. Le tableau 6.5 présente les résultats obtenus. En l'analysant nous constatons que :

- le rajout de paires de *threads* communicants augmente le débit total.
- la taille de messages pour le mi-débit($n_{\frac{1}{2}}$) a été réduite. Ceci signifie que l'on atteint plus vite un régime soutenu de communications par rapport au cas d'un seul *thread*. Nous attribuons cela à un recouvrement des communications entre elles. Pendant qu'un *thread* attend la réponse à un message, un autre *thread* peut envoyer son message.
- le recouvrement se traduit par un temps d'amorçage pour n paires inférieur à n fois celui pour 1 paire.

Par rapport aux deux version d'ATHAPASCAN-0, on notera une différence de performance en faveur d'ATHAPASCAN-SMP. En particulier, il résiste mieux à la surcharge d'accès à MPI. Cette caractéristique est entièrement due à la reconception du verrouillage d'accès et l'emploi de l'attente active.

Le recouvrement de communications par des calculs

Afin de mesurer le recouvrement de communications par des calculs offert par ATHAPASCAN-0, nous avons pris le jeu de test POLY3 présenté à la section 3.5.3. Comme pour le test *ping-pong*, ce test met en jeu plusieurs paires de *threads* (*ping* et *pong*). A la différence du test précédent le *thread pong* doit effectuer un calcul sur les données reçues et renvoie le résultat au *thread ping*. Le calcul effectué par notre test est l'évaluation d'un polynome de degré 2 ($p(x) = a.x^2 + b.x + c$) par la règle de Horner. Ce polynome est évalué pour un ensemble de valeurs x correspondant à

[7]Le nombre de paires est un paramètre d'exécution.

Tableau 6.5 *Ping-pong multiples : les indicateurs r_∞,t_0 ,$n_{\frac{1}{2}}$ - protocole* short
(Biprocesseur, réseau Ethernet 100 Mpbs, LAM version 6.3, mode c2c)

Débit(r_∞^{64k})	1 paire	2 paires	4 paires	8 paires
ATHAPASCAN-0	7.80 Mo/s	8.0 Mo/s	8.0 Mo/s	8.0 Mo/s
ATHAPASCAN-SMP	7.90 Mo/s	8.0 Mo/s	8.20 Mo/s	8.20 Mo/s

Mi-débit($n_{\frac{1}{2}}$)	1 paire	2 paires	4 paires	8 paires
ATHAPASCAN-0	≈4.75 Ko	≈3 Ko	≈2.5 Ko	≈2 Ko
ATHAPASCAN-SMP	≈4 Ko	≈2 Ko	≈1.75 Ko	≈1.5 Ko

Amorçage(t_0)	1 paire	2 paires	4 paires	8 paires
ATHAPASCAN-0	495μs($\pm$6)	840μs($\pm$14)	1832μs($\pm$50)	2962μs($\pm$64)
ATHAPASCAN-SMP	335μs($\pm$1.8)	563μs($\pm$5)	1097μs($\pm$14)	2170μs($\pm$11)

des éléments d'un tableau de nombres flottants. On fait varier la taille de ce tableau de zéro à 16384 (ce qui correspond à des messages de zéro à 64k octets).

Tableau 6.6 *Exécution de POLY3 : les indicateurs r_∞, $\tilde{r}_\infty$,$f_{\frac{1}{2}}$ - protocole* short *LAM*
(Biprocesseur, réseau Ethernet 100 Mpbs, LAM version 6.3, mode c2c)

Débit(r_∞^{64k})	1 paire	2 paires	4 paires	8 paires
ATHAPASCAN-0	3.36 Mo/s	4.70 Mo/s	6.42 Mo/s	6.9 Mo/s
ATHAPASCAN-SMP	3.35 Mo/s	4.68 Mo/s	6.35 Mo/s	6.75 Mo/s

Débit($\tilde{r}_\infty^{64k}$)	1 paire	2 paires	4 paires	8 paires
ATHAPASCAN-0	1.77 MFlops	2.45 MFlops	3.37 MFlops	3.65 MFlops
ATHAPASCAN-SMP	1.76 MFlops	2.45 MFlops	3.32 MFlops	3.55 MFlops

Demi-intensité($f_{\frac{1}{2}}$)	1 paire	2 paires	4 paires	8 paires
ATHAPASCAN-0	≈576 floats	≈480 floats	≈480 floats	≈480 floats
ATHAPASCAN-SMP	≈448 floats	≈384 floats	≈352 floats	≈352 floats

Le test a été fait pour 1,2,4 et 8 paires de *threads*. Le tableau 6.6 présente les résultats obtenus. On mesure le temps total des *ping-pongs* et du calcul. A partir de ce temps, on obtient la quantité de données transferées par seconde (débit) et le nombre d'opérations en flottants réalisées. On constate :

- une augmentation du nombre d'opérations flottantes à mesure qu'on augmente le nombre de paires de *threads*. Ceci prouve un recouvrement de communications par des calculs. Nous avons observé une croissance sensible du nombre d'opérations réalisées jusqu'à 8 paires de *threads*. Jusqu'à 16 paires cette croissance est peu sensible. A partir de 24 paires, la performance se dégrade. Cette dégradation est due au surcoût de gestion des *threads*, une fois

atteint le débit de calcul maximum.

- une supériorité de la version SMP sur la version originale pour des tableaux inférieurs à environ 6184 flottants. Ceci peut être observé sur l'indicateur $f_{\frac{1}{2}}$: plus il est petit, moins nous avons de surcoût de communications. Par conséquent, le processeur est libre pour réaliser plus de calcul. On remarquera aussi que la version SMP atteint son asymptote ($\bar{r}_\infty$) avant la version originale.
- la version SMP donne des résultats moins bons pour les asymptotes r_∞ et $\bar{r}_\infty$ que la version originale. Cette différence augmente avec le nombre de paires de *threads*.

Nous expliquons ce dernier point par un effet collatéral de la technique de *spin-lock*. En effet, en augmentant le nombre de paires de *threads* communicants, nous augmentons la probabilité d'un conflit sur un des verrous de la gestion de requêtes. Eventuellement, un *thread* peut perdre le processeur en ayant pris ce verrou ; ainsi les autres *threads* (et le propre démon de communication) exécuteront la boucle d'attente active pour ce verrou (*spin-lock*) jusqu'à sa fin. Cette situation persistera jusqu'à ce que le verrou soit libéré, d'où la perte d'efficacité. Plus important est le temps du *spin lock*, pire sont les résultats.Il faut donc réduire le nombre d'itérations de test avant le blocage. A la limite, éliminer le test de *spin lock* nous amène à la version originale d'ATHAPASCAN-0. Pour confirmer cette hypothèse nous avons diminué le nombre d'itérations de test de *spin lock* avant blocage. Nous avons obtenu une amélioration du comportement d'ATHAPASCAN-SMP.

Bilan partiel

Nous avons analysé le comportement d'ATHAPASCAN-0/SMP lorsqu'on utilise de façon combinée la multiprogrammation légère et les communications. Nous avons pu vérifier que la multiprogrammation légère permet le recouvrement des communications entre elles ainsi que par des calculs. On a soulevé un problème de performance lié au réglage des *spin lock*. On a constaté une meilleure performance d'ATHAPASCAN-SMP par rapport à ATHAPASCAN-0.

Toutefois dans les expériences précédentes, nous n'avons cherché à évaluer que des points isolés du comportement d'un noeud, comme l'exploitation du parallélisme SMP ou le recouvrement de calcul et des communications. Comment se comporte ATHAPASCAN-0 en présence de charge de calcul et de communications interdépendentes ? Essayer de répondre à cette question est l'objectif de la section suivante.

6.3 Analyse du comportement global

Dans cette section, nous allons essayer de caractériser le comportement d'ATHAPASCAN-0 d'une façon globale en effectuant des tests basés sur des situations plus « réelles ». Les tests réalisés jusqu'à lá essaient de mettre en évidence une caractéristique particulière. Ici nous nous préoccuperons d'analyser l'ensemble. Initialement nous rappelons le problème du compromis efficacité/réactivité. Ensuite à partir d'une charge dont nous ferons varier la découpe, nous essaierons d'identifier des régimes de fonctionnement relativement à cette question d'efficacité et réactivité. Pour conclure nous étudierons le comportement sous charge.

6.3.1 L'ordonnancement, la réactivité, et l'efficacité

Un compromis est nécessaire entre réactivité et l'efficacité dès qu'**aucun mécanisme** ne permet de déclencher la scrutation au bon moment. Les solutions logicielles, *thread* spécialisé ou à la demande coûtent des exécutions inutiles. Réaliser la scrutation avec une fréquence trop élevée augmente la réactivité et diminue l'efficacité du fait de tests inutiles. Le choix de la période de scrutation peut être laissé au programmeur de l'application ou assuré par le noyau. Dans le cas d'une scrutation par un démon spécialisé, la période de scrutation va donc dépendre de l'ordonnancement des *threads*. Il en est de même de la dégradation du service de calcul.

Nous avons experimenté, pour le cas où la scrutation est assurée par un *thread* spécialisé, deux types d'ordonnanceurs. Dans la version initiale d'ATHAPASCAN-0, ATHAPASCAN-0b[86], sur SPx (DCE *threads*, modèle N :1), l'ordonnanceur était du type FIFO à priorité fixe. Dans la version ATHAPASCAN-0 et ATHAPASCAN-SMP, l'ordonnanceur (POSIX *threads*, modèle 1 :1) est du type UNIX à partage équitable (priorités équitables).

Ordonnanceur à priorité : En présence de tels ordonnanceurs dans cette hypothèse, trois niveaux de priorités sont utilisés : le *thread* spécialisé étant le plus prioritaire, les *threads* de calcul ayant des niveaux des priorités intermédiaires, et un *thread* chien de garde avec une priorité plus basse. Aucun moyen n'existe pour bloquer directement un *thread* de calcul en attente d'arrivée d'un message sur le réseau. L'interaction entre les *threads* est la suivante :

1. Le *thread* spécialisé se bloque sur une variable de condition en attendant que les *threads* de calcul réalisent une requête de communication.
2. Le depôt de la requête réveille le *thread* spécialisé pour qu'il achemine la requête et reçoive les messages. Il se bloque à nouveau.
3. Dès que tous les *threads* de calculs sont bloqués en attente d'une synchronisation, ou de la réception d'un message, le *thread* moins prioritaire (chien de garde) s'exécute et réveille le *thread* spécialisé. Celui-ci tourne de façon

continue pour assurer la progression des communications. Lorsqu'il réveille un *thread* de calcul en attente de communication, il s'endort à nouveau.

Ce schéma présente deux problèmes. Primo, si les *threads* de calcul réalisent plus de calcul que de communications, on retarde la détection de messages. La réactivité est mise en cause. Secundo, si les *threads* de calcul sont bloqués sur des synchronisations locales, et non sur des communications, le chien de garde se déclenche et le *thread* spécialisé empêche l'exécution des *threads* de calcul. Ici c'est l'efficacité qui est mise en cause.

Dans cette solution, le régime de fonctionnement du démon est donné par le flux de requêtes de communication qui dépend de l'application. Si ce flux est mal accordé au comportement du réseau, les communications sont assurées par le démon lorsque les *threads* se bloquent. Nous n'avons pas utilisé cette version suffisament pour la caractériser.

Un autre example d'ordonnanceur à priorité est celui du noyau MARCEL (section 4.3.5). Ici, augmenter la priorité d'un *thread* veut dire tourner plus longtemps, la diminuer tourner moins longtemps. Chaque *thread* utilise un pourcentage du temps processeur en fonction de sa priorité : $(p(k_i)/\sum_{j=1}^{n} p(k_j)$. La fréquence de scrutation la plus rapide est quand tous les *threads* ont une priorité de 1 : on scrute tous les $k \times q$ pour une durée q, où q est le *quantum* d'exécution. Si on augmente la priorité du *thread* de scrutation à x, il scrute tous les $((k-1)+x) \times q$ pendant $n \times q$. Si on augmente la priorité des autres *threads* à une valeur x, on scrute tous les $(k-1) \times x \times q + q$ pendant une durée q. La réactivité est fortement dépendente du nombre de *threads* et de leurs respectives priorités. Il est pratiquement impossible de fixer des bornes pour la scrutation, par conséquent cette méthode de scrutation n'est pas appropriée. A cause de cela PM^2 l'intègre dans MARCEL (fait à chaque commutation de contexte).

Ordonnancement équitable : les versions ATHAPASCAN-0 et ATHAPASCAN-SMP tournent sur un noyau POSIX de type 1 :1 qui effectue un ordonnancement équitable du point de vue du temps de calcul attribué aux *threads*. L'ordonnanceur disponible est normalement un ordonnanceur de type UNIX, c'est à dire préemptif à priorité dynamique[8]. La seule précision disponible indique une attribution équitable du temps de calcul aux *threads*. Le surcoût de la scrutation est estimé par $1/(1+k)$ en présence de k *threads* de calcul. Sur un tel ordonnanceur l'influence de la scrutation diminue avec le nombre de *threads*. On peut considérer qu'en moyenne la scrutation tourne avec un période de $(k+1) \times q$. En d'autres termes on réduit la réactivité de la scrutation. La durée maximale d'une phase de scrutation est bornée par le *quantum*. Néanmoins, en réalité, l'exécution des *threads* est fonction de leur priorités dynamiques. Comme il ne nous est pas apparu de moyen pour contrôler ces paramètres nous avons fourni au programmeur la possibilité de forcer le lance-

[8]Certains noyaux systèmes comme Solaris 2.6 et IRIX 6.5 offrent les politiques SCHED_FIFO et SCHED_RR pour leur noyaux 1 :1, toutefois il faut exécuter l'application avec des privilèges de super-utilisateur. Comme ce n'est pas le cas de la « population cible » d'ATHAPASCAN nous avons écarté cette hypothèse de travail.

ment d'une scrutation par un opérateur (primitive *a0Advance()*). Les expériences confirment cette prédiction (cf. section 5.3). Cette influence est plus visible sur des monoprocesseurs (fig. 5.3).

Un autre choix que nous avons fait est qu'à chaque fois que le démon réveille un *thread* en attente, ou qu'il a parcouru la liste de requêtes complètement, il essaye de rendre la main en réalisant des appels à une primitive type (*yield()*). Ceci est fait dans le but d'éviter (potentiellement) que le démon tourne pendant tout son *quantum*. Dans ce cas on réduit l'influence du « vol » de temps du démon, et comme il n'utilise pas tout son *quantum* le système tend à lui attribuer une priorité plus élevée que s'il l'utilisait complètement. L'ordonnanceur système garanti l'exécution des autres *threads*. Ce choix a été approprié sur des expérimentations sur nos plate-formes (Solaris 2.6).

Si l'on peut caractériser effectivement le coût de la scrutation (section 5.3), il est difficile de voir l'incidence de ce fonctionnement sur une application parallèle (réactivité). Les sections suivantes tentent d'apporter une réponse à cette question.

6.3.2 La découpe calcul-communication

Le principe général de parallélisation d'un calcul est de le découper en sous calculs. Ces sous calculs interagissent entre eux de façon plus ou moins structurée. Dans un environnement distribué cette parallélisation se fait donc au prix d'une charge de communication. Plus le découpage s'affine, plus la charge de communication croît. Quelques schémas simples existent. Le premier est celui de la ferme de processus. Une tâche maître (fig. 6.6.a) distribue du travail à des tâches esclaves qui renvoient les résultats. Ce schéma est approprié à des calculs élémentaires indépendants ; les tâches esclaves ne communiquent pas entre elles. Dans un autre schéma (fig. 6.6.b), les tâches esclaves enchaînent des étapes de calcul et des étapes d'échange de données. Ces échanges dépendent complètement de l'application.

Notre propos est d'essayer de voir l'incidence de la charge de calcul et de communication sur le fonctionnement d'ATHAPASCAN-0. Il nous faut donc une application type où nous pouvons faire varier de façon contrôlée la découpe calcul-communication. Notre choix s'est donc porté sur un modèle d'application de type ferme de processus. Pour le calcul à paralléliser, nous avons repris le calcul de Mandelbrot dans sa version adaptée (Mandelbrot synthétique ou régulier). L'expérience consiste à exécuter Mandelbrot pour un plan 1024x1024 sur deux noeuds biprocesseurs, un jouant le rôle de maître et l'autre d'esclave. Sur l'esclave, on crée 1, 2, 4, 8 ou 16 *threads*. Le maître envoie des régions à calculer à chacun de ces *threads*. Lorsqu'un *thread* a fini le calcul d'une région, il renvoie le résultat vers le maître. Celui-ci, tant qu'il y a du calcul, renvoie au *thread* une nouvelle région. Pour avoir une référence de comparaison nous avons exécuté aussi une version MPI « seul » de Mandelbrot aussi basée sur une ferme de processus.

En observant les courbes de la figure 6.7 on constate qu'elles présentent une

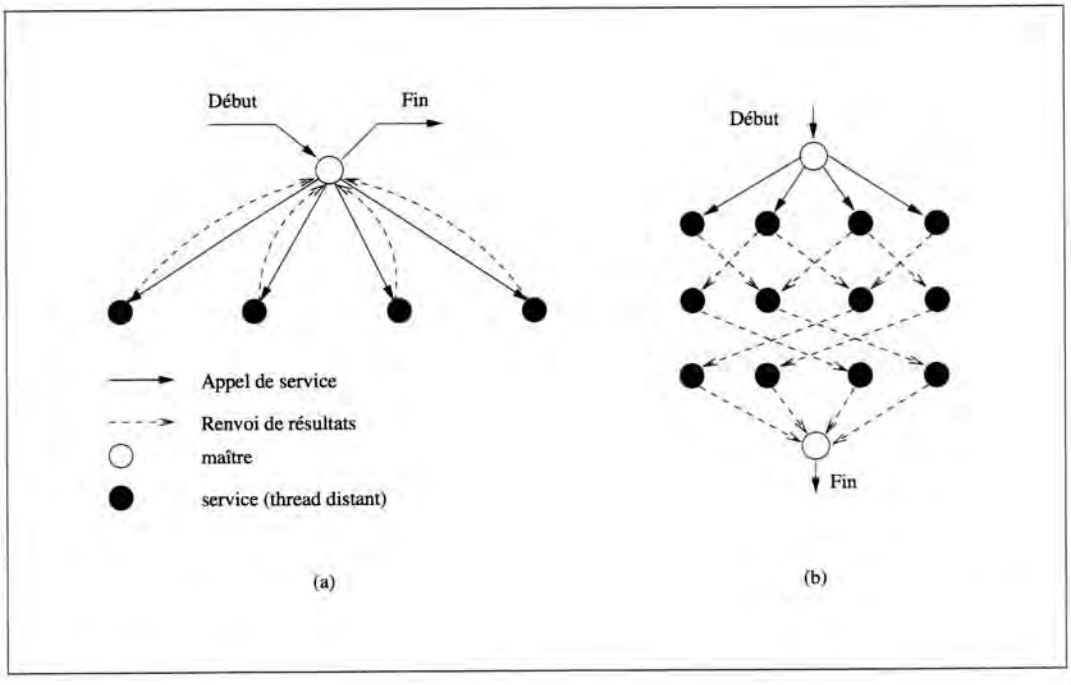

Figure 6.6 *Schémas de découpe de données*

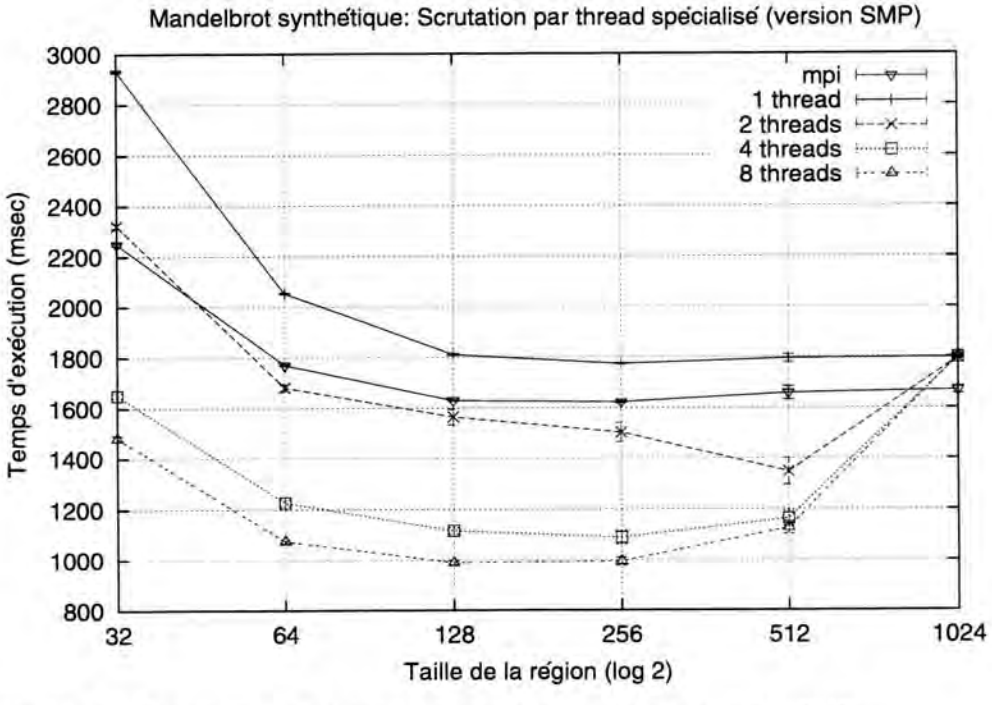

Figure 6.7 *Évaluation de Mandelbrot (synthétique) (*ATHAPASCAN-*SMP) (Biprocesseur, réseau Ethernet 100 Mpbs, mode -c2c)*

courbure en forme de « selle de cheval » pour un nombre de *threads* supérieur à 2. Cette courbure est en partie expliquée par l'existence de plus d'un processeur

physique. Dans notre cas, il s'agit des biprocesseurs. On peut attendre une accélération théorique d'un facteur 2. Ceci n'est pas le cas pour toutes les découpes :

- pour une région 1024x1024, nous avons 1 seule région à calculer correspondant à la taille de l'image. Indépendamment du nombre de *threads*, un seul réalise tout le calcul. Ceci explique le fait que toutes les courbes se rejoignent. On observe aussi un écart entre la courbe MPI et les courbes ATHAPASCAN. Nous l'attribuons à l'acheminement des messages par le démon de communication qui n'est pas présent dans la version MPI.
- pour une découpe en régions de 512x512 nous avons 4 régions. On ne gagne pas en performance en rajoutant plus de 4 *threads*. En considérant que ce test a été fait sur un biprocesseur, on pourrait s'attendre à une accélération similaire pour les cas 2, 4 et 8 *threads*. Comme il y a 4 régions seulement 4 *threads* travaillent effectivement il est normal que le résultat soit le même que pour 4 *threads*. La différence par rapport au cas 2 *threads* est expliquée par l'influence du démon ($3/4T$: 2 *threads* contre $5/8T$: 4 *threads*).
- pour une découpe en région 256x256 nous avons 16 régions donc assez de travail pour 1, 2, 4 et 8 *threads*. On notera deux choses. La différence entre le gain obtenu entre 2, 4 et 8 *threads* est due en partie au « vol » de temps de calcul fait par le démon. D'autre part, en considérant la granularité calcul-communication, 2 *threads* ne génèrent pas assez de parallélisme. La deuxième chose à noter est que le gain pour la découpe 512x512 est légèrement inférieur que celui obtenu pour la découpe 256x256 pour 4 et 8 *threads* (visible sur la courbe 4 *threads*). Cette différence est expliquée par un recouvrement des communications par le calcul. La découpe 256x256 offre 4 fois moins de poids de calcul et 4 fois plus de messages que la découpe 512. Les *threads* de calcul exécutent moins de temps avant d'envoyer un message. Cet envoi se superpose au calcul d'une autre région. Avec la découpe 512x512 les *threads* calculent plus longtemps avant d'envoyer un résultat.
- pour une découpe en régions 128x128 on augmente le nombre de régions par un facteur 4 (par rapport à la découpe 256x256) et on diminue le calcul aussi par un facteur 4. Le nombre de communications reste constant par région ce qui provoque une augmentation du nombre de communications et de leur fréquence car le temps de calcul est réduit. Les communications deviennent un goulot d'étranglement. C'est à nouveau un problème de granularité calcul-communication. Il en est de même pour la découpe 64 et 32. Ce phénomène est observable aussi sur l'exécution MPI « seul ».

Pour vérifier que notre analyse est toujours valable pour un cas « réel », nous avons exécuté exactement le même programme, avec les mêmes paramètres pour un « vrai » Mandelbrot (en effet le choix entre un « vrai » Mandelbrot et le « synthétique ou régulier » est une option de lancement). Le résultat obtenu est présenté par la figure 6.8, où on observe un comportement similaire à celui analysé. La principale différence est que le Mandelbrot réel est irrégulier et les régions présentent

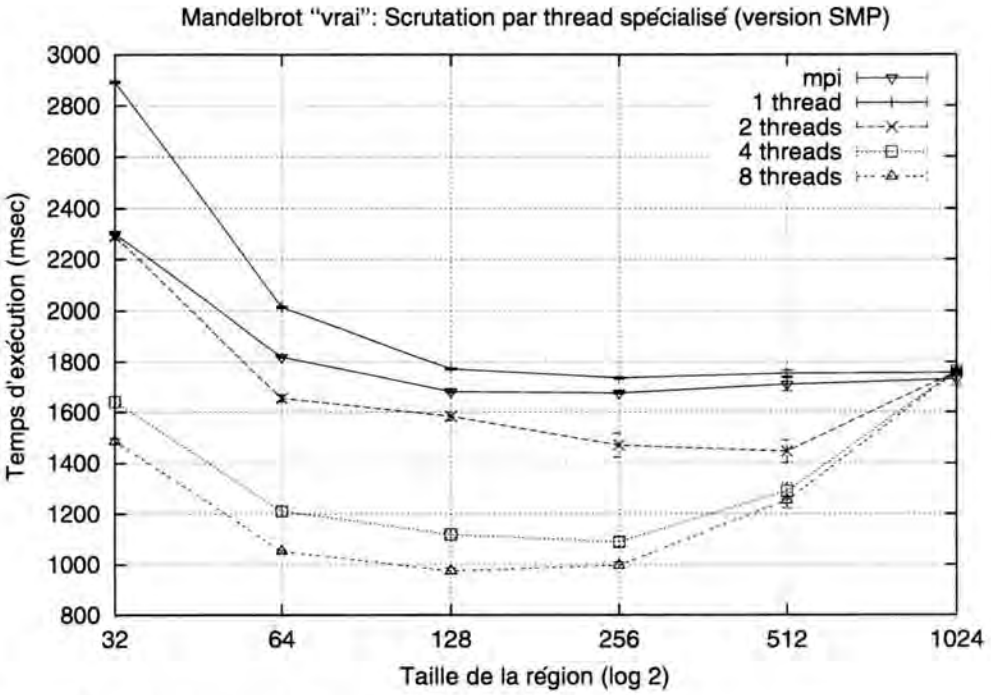

Figure 6.8 *Évaluation de Mandelbrot (le « vrai ») (*ATHAPASCAN-*SMP) (Biprocesseur, réseau Ethernet 100 Mpbs, mode -c2c)*

un poids de calcul déséquilibré.

Enfin pour montrer la meilleure gestion de communications d'ATHAPASCAN-SMP par rapport à la version originale, nous avons reexécuté le programme de Mandelbrot (le « vrai ») en utilisant ATHAPASCAN-0. La figure 6.9 donne le résultat de cette exécution. On remarque une grande différence d'efficacité entre les deux versions d'ATHAPASCAN. Cette écart est plus important pour de petites régions (taille 32). Cet écart se réduit à mesure que la taille des régions augmente. Ceci s'explique par le fait que plus les régions sont petites, plus des messages sont générés. Par conséquent on augmente le conflit d'accès aux structures partagées (liste de requêtes) entre le démon et les *threads* de calcul. Cette situation fait ressortir les avantages de la nouvelle gestion de requêtes proposée par ATHAPASCAN-SMP.

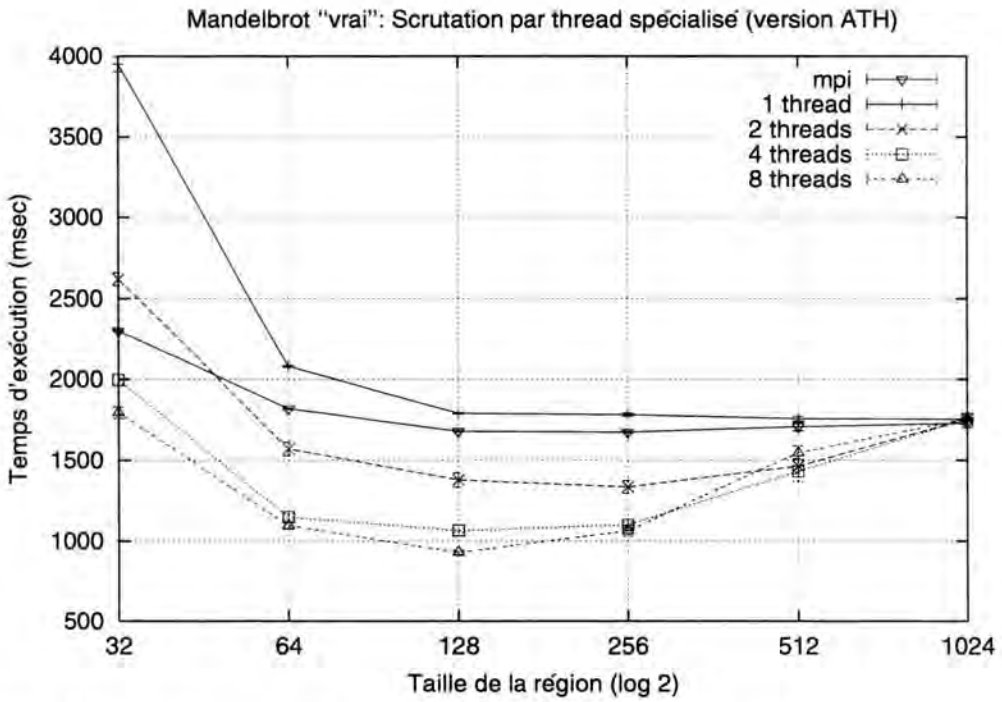

Figure 6.9 *Évaluation de Mandelbrot (le « vrai ») (*ATHAPASCAN-*0) (Biprocesseur, réseau Ethernet 100 Mpbs, mode -c2c)*

6.4 ATHAPASCAN-SMP et quelques applications APACHE

Le noyau exécutif ATHAPASCAN-0 a été conçu pour permettre le développement d'applications scientifiques et pour servir comme support exécutif à ATHAPASCAN-1. Dans cette section nous allons donc évaluer le comportement d'ATHAPASCAN-SMP par rapport à ces objectifs originaux. Pour cela nous comparerons la performance obtenue par l'exécution d'applications scientifiques écrites en ATHAPASCAN-0 et en ATHAPASCAN-1 en utilisant comme couches de base ATHAPASCAN-SMP et ATHAPASCAN-0. Les mesures que nous présenterons par la suite ont été réalisées en utilisant les applications développées au sein de l'équipe APACHE par T. Gautier, E. Weibel, P.E. Bernard (dynamique moléculaire), A. Charão (équations dérivées partielles), G. Cavalheiro (Fibonacci et Jacobi), et M. Doreille (Gauss).

L'intérêt majeur de ces applications est qu'elles utilisent en conditions « réelles » différentes primitives ATHAPASCAN-0/SMP. En plus, leurs rapports calcul-communication sont différents. Toutes les exécutions ont été faites en utilisant comme méthode de scrutation celle « par *thread* spécialisé (démon de communication) ».

6.4.1 Takakaw : Dynamique moléculaire

Takakaw est une application parallèle de simulation de systèmes moléculaires complexes développée initialement par P.E. Bernard au cours de sa thèse [15] et actuellement reprise dans le cadre de l'action incitative INRIA (SIMBIO[9]). L'étude de la dynamique moléculaire classique consiste à réaliser une simulation du mouvement des atomes et des molécules d'un système donné en utilisant les équations traditionnelles de la mécanique des solides (équations de Newton et de Langevin).

Le calcul du mouvement de l'ensemble de particules (atomes) est un problème à N-corps. Pour le calculer on effectue itérativement une intégration numérique des équations de mouvement. Ainsi, à partir des positions et vitesses initiales des atomes, données d'entrée du problème, on calcule la suite au cours du temps des positions et des vitesses des atomes du système. Dans le modèle utilisé traditionnellement en biologie, trois types de forces sont considérés : les forces non liées (électrostatiques) ; les forces des interactions géométriques ; et les forces de contraintes. L'algorithme général est constitué par une boucle de calcul où on évalue l'ensemble de ces forces, et ensuite on les combine pour trouver les nouvelles positions des atomes.

La méthode employée par Takakaw est celle du rayon de coupure (*cutt off*). On calcule seulement les interactions entres atomes se trouvant à une distance inférieure à une valeur donnée R_c. L'algorithme implantant cette méthode est basé sur un découpage géométrique du système à simuler. Ici l'espace de simulation est divisé en boîtes cubiques dont la taille dépend de R_c. Ainsi implicitement les atomes voisins d'un atome donné se trouvent dans la boîte de l'atome considéré et dans les 26 boîtes voisines. Si l'on considère une boîte par processus « lourd », chacun des processus doit échanger la position des atomes de sa boîte avec les 26 processus possèdant des boîtes voisines. Après cette étape, chaque processus est donc capable de calculer les forces d'interactions. Ces processus correspondent à des tâches ATHAPASCAN-0. Takakaw réalise un placement statique de boîtes sur les tâches ATHAPASCAN-0 selon un critère de distribution équitable des atomes par tâche. Par conséquent, le nombre de boîtes sur chaque tâche peut varier. A l'intérieur de chaque tâche, des *threads* s'occupent du calcul des forces et réalisent les communications pour l'ensemble de boîtes.

Nous avons donc exécuté Takakaw en utilisant trois molécules différentes : une « petite » (3625 atomes), une « moyenne » (11615 atomes) et une « grande » (67598 atomes). Le rapport calcul-communication varie selon la taille de la molécule : le calcul croit avec le carré du nombre d'atomes et les communications croissent linéairement. Une autre caractéristique de cet algorithme est qu'il évalue l'intégration des équations de mouvement en effectuant un nombre n d'itérations. Les premières itérations sont marquées par un mouvement plus important des atomes, ce qui implique plus de communications à la phase initiale car les atomes peuvent changer de « boîtes ». Une situation plus au moins stationnaire est atteinte ensuite,

[9]http ://altair.iecn.u-nancy.fr/~bernard/SIMBIO

et les communications sont réduites. P.E. Bernard dans son travail de thèse employait 50 à 100 itérations. Nous en avons utilisé 50. Le tableau 6.7 présente les temps d'exécution mesurés pour ces trois molécules.

Tableau 6.7 *Temps d'exécution Takakaw*
(Biprocesseurs, solaris 2.6, réseau ethernet 100 Mbps)

2 tâches	3625 atomes	11615 atomes	67598 atomes
ATHAPASCAN-0	1.465±0.015s	4.890±0.042s	42.46±0.95s
ATHAPASCAN-SMP	1.282±0.002s	4.605±0.006s	41.55±0.14s
4 tâches	3625 atomes	11615 atomes	67598 atomes
ATHAPASCAN-0	0.784±0.005s	2.75±0.01s	24.86±0.35s
ATHAPASCAN-SMP	0.722±0.01s	2.57±0.02s	23.94±0.06s
8 tâches	3625 atomes	11615 atomes	67598 atomes
ATHAPASCAN-0	0.88±0.017s	2.474±0.018s	24.52±0.21s
ATHAPASCAN-SMP	0.72±0.01s	2.36±0.003s	23.43±0.09s

L'application Takakaw a été conçue dès le départ pour être exécutée sur un environnement distribué avec au moins deux tâches. Comme nous l'avons récupérée sans la modifier nous n'avons pas d'estimation du temps séquentiel (T_s) ou du temps d'exécution du code parallèle sur un seul noeud (T_1). On peut simplement observer que la version SMP offre toujours de meilleurs résultats que la version originale. On constate aussi qu'à partir de 4 tâches (2 par noeud) on exploite le parallélisme réel offert par les biprocesseurs. Dans ce cas, le temps d'exécution est environ la moitié de celui avec 2 tâches (1 par noeud). On observe encore qu'à mesure que la granularité augmente, la différence entre les deux versions se réduit. Cet effet nous paraît logique car nous sommes en train de réduire l'accès aux structures partagées entre l'application (dépôt des requêtes) et le démon de communication. On réduit ainsi les problèmes de conflits sur les verrous.

6.4.2 AHPIK : Équations dérivées partielles

AHPIK [40][39] est un environnement modulaire qui permet le développement de solveurs parallèles pour la résolution d'équations aux dérivées partielles par une méthode de décomposition de domaine. AHPIK s'appuie sur des concepts de programmation orientée objet pour construire un noyau parallèle - basé sur la multiprogrammation légère - commun aux différentes méthodes de décomposition de domaine. Pour atteindre ces objectifs AHPIK est programmé en C++ et utilise ATHAPASCAN-0 comme noyau exécutif. La conception et l'évaluation de cet environnement constitue le travail de thèse actuellement en cours de Andrea Charão dans le projet APACHE.

Le solveur parallèle est composé d'une ou plusieurs tâches[10] responsables du calcul de la solution sur un ensemble de sous-domaines. A l'initialisation, chaque tâche lit un schéma de placement global des sous-domaines. Ensuite pour chaque sous-domaine, trois *threads* sont créés :

- un *thread* (T_Ω) de calcul responsable de l'évaluation des calculs locaux relatifs au sous-domaine. Ce *thread* envoie aussi une erreur locale (critère de convergence) à un *thread* maître à la fin de chaque interaction (envoi asynchrone) ;
- deux *threads* (T_γ) qui s'occupent des calculs des données frontalières et des envois/réceptions de messages aux sous-domaines voisins (interface) ;

Le calcul suit un schéma itératif synchrone en alternant calculs locaux et calculs d'interface. La convergence du processus itératif dépend du calcul de l'erreur globale effectué par un *thread* maître placé dans un noeud dédié ou non. Le maître ne communique avec les esclaves que pour signaler la convergence et déclencher la terminaison du programme parallèle (dans chaque tâche il y a encore un *thread* qui s'occupe de recevoir le message de terminaison et arrêter les autres *threads*).

Nous présenterons donc les résultats (en temps d'exécution) obtenus par l'exécution d'une application AHPIK sur les deux versions d'ATHAPASCAN-0. On cherche la solution des équations de convection-diffusion qui sont une version linéarisée des équations de Navier-Stokes. A ce titre, elles sont utilisées dans de nombreux problèmes de mécaniques de fluides. Parmi eux, des problèmes d'environnements tels que le transport et la diffusion de polluants dans l'athmosphère ou l'océan, de sédiment dans les fleuves, etc.

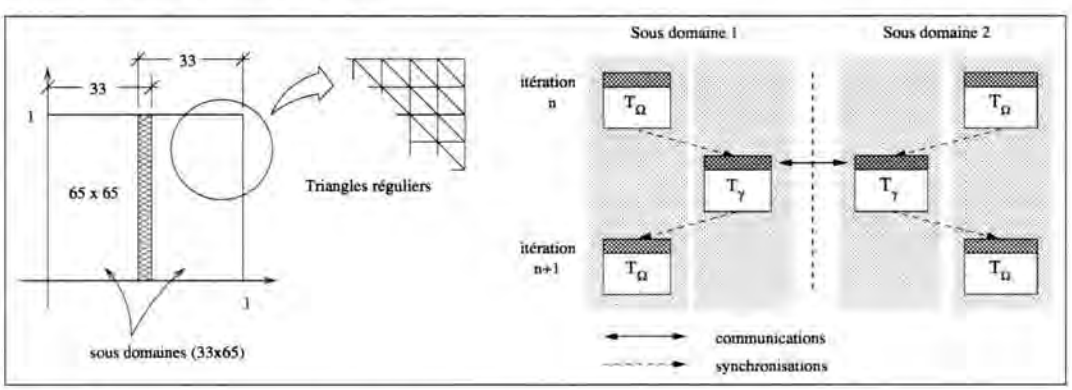

Figure 6.10 *Schéma parallèle pour une décomposition en 2 sous-domaines*

Dans l'exécution réalisée, le domaine de calcul utilisé est le carré unitaire donné par $\Omega =]0,1[\times]0,1[$, discrétisé en triangles inscrits dans une grille régulière 65×65, ce qui conduit à un système à 4225 inconnues. Il est décomposé en 2 sous-domaines 65×33 chacun (il y a une colonne superposée). La figure 6.10 schématise cette situation. Les 2 sous-domaines ont été placés soit tous sur une seule tâche ATHAPAS-

[10]Tâches dans le sense ATHAPASCAN-0 c'est à dire des processus lourds.

CAN-0 soit chacun sur un tâche ATHAPASCAN-0, en utilisant 1 à trois machines différentes (noeuds physiques). Dans ce dernier cas, le *thread* maître tourne sur un noeud dédié. On notera que l'exécution effectuée est considérée comme « petite ». Le nombre d'itérations est déterminé par un critère de convergence, et la phase de calcul s'accélère à mesure que la solution approche de la convergence (56 itérations pour ce cas). Les messages échangés entre les *threads* T_γ ont une taille d'environ 256 octets. Il s'agit d'une application où le temps de calcul prédomine sur la communication. Par contre, les phases de calculs sont synchronisées à la fin de chaque itération par des messages de mise à jour des données frontalières. Le tableau 6.8 présente les résultats obtenus.

Tableau 6.8 *Temps d'exécution AHPIK pour une décomposition en 2 sous-domaines (1 machine = biprocesseur ; 3 machines = 2 biprocesseurs, 1 monoprocesseur (maître))*

	1 tâche 1 machine	2 tâches 1 machines	2 tâches 3 machines
ATHAPASCAN-0	49.28 s	89.53 s	55.6 s
ATHAPASCAN-SMP	46.46 s	79.33 s	30.6 s

Exécution séquentiel T_s : 60.49 s

Analysons initialement le cas où les 2 sous-domaines sont placées sur une seule tâche ATHAPASCAN-0. À l'intérieur d'un sous-domaine les *threads* T_Ω et T_γ s'exécutent l'un après l'autre. Comme le temps de calcul de T_Ω est beaucoup plus important que T_γ on peut supposer que le temps total est donné par T_Ω. L'application étant régulière, il est raisonnable aussi de considérer que la charge de travail est divisée équitablement entre les *threads* T_Ω. En supposant que le démon de communication tourne pendant le même temps que les *threads* de calcul nous pouvons écrire $T_t = T_c + T_d = T_c + T_c/2 = 3/2T_c$. Comme nous sommes en présence d'un biprocesseur T_t est divisé par deux et devient $3/4T_c$. Le temps séquentiel (T_s) obtenu par l'exécution de cette application est de $\approx$ 60.5s, par conséquent on peut estimer que l'exécution parallèle prendra $\approx$ 45.37 ($3/4T_c$). Nous observpns 46.46s (SMP) et 49.28s. Ces résultats corroborent notre hypothèse du vol de temps d'exécution du démon de communication.

Dans la situation où 2 tâches sont sur un seul noeud, nous sommes dans un cas de figure similaire au précédent à l'exception près que nous avons deux démons de communications. Ici le temps total de calcul est donc estimé par $T_t = T_c + 2T_d = 2T_c$. En présence d'un biprocesseur T_t devient égal à T_c. Les temps mesurés correspondent à 79.33s (SMP) et 89.53s. Nous expliquons cet écart entre les valeurs prévues et celles observées par les synchronisations nécessaires entre les tâches pour communiquer.

En plaçant un sous-domaine par tâche et en exécutant les tâches sur des noeuds physiques différents, nous avons sur chaque noeud un démon de communication et

un *thread* T_Ω. Globalement nous avons 4 *threads* (2 démons et 2 T_Ω) pour s'exécuter sur 4 processeurs. Le temps d'exécution T_t est estimé par $T_t = T_c + 2T_d = 2T_c$ qui divisé par 4 processeurs nous donne une estimation de 30.25s contre 30.65 (SMP) et 55.6s mesurés.

De cela on peut conclure deux choses. La première est la confirmation de notre modèle de coût du démon de scrutation. La deuxième est que le fait d'éviter les commutations de contexte par l'usage de *spin lock* permet au démon de communication tourner au même temps que le *thread* T_γ (responsable pour la mise à jour des interfaces). Ainsi sur la version originale, le dépôt d'une requête implique des blocages du démon c'est à dire des commutations de contexte. Celles-ci sont évitées par la version SMP d'où le gain de performance.

6.4.3 ATHAPASCAN-1

ATHAPASCAN-1 constitue l'interface applicative de l'environnement ATHAPASCAN. Il offre un modèle de programmation qui permet la description du parallélisme de l'application de façon indépendante de la machine cible. Basé sur un graphe de flot de données, ce modèle garantit une sémantique d'accès aux données deterministe avec un modèle de coût associé. ATHAPASCAN-1 assure aussi un ordonnancement applicatif avec régulation dynamique de charge. Les problèmes scientifiques abordés par l'équipe ATHAPASCAN-1 sont l'ordonnancement applicatif, la gestion répartie du flot de données (graphe de tâches), et l'adéquation du modèle proposé au calcul scientifique à caractéristiques irregulières (par exemple, Cholesky).

L'exécution d'un programme ATHAPASCAN-1 implique la création d'une machine virtuelle. Cette machine virtuelle est composée par un ensemble de noeuds (tâches ATHAPASCAN-0 c'est à dire des processus « lourds ») ; sur chaque noeud est créé un pool d'exécution (*a1_pool*) constitué par un certain nombre de processeurs virtuels (*threads* ATHAPASCAN-0) dédiés à l'exécution de tâches[11]. L'application est vue comme un graphe de tâches ; un ordonnanceur applicatif attribue ces tâches aux processeurs virtuels de façon complètement transparente à l'utilisateur. Une description plus complète de ce mécanisme d'ATHAPASCAN-1 est donné par G. Cavalheiro dans son travail de thèse[35].

ATHAPASCAN-1 est conçu en utilisant comme noyau de base ATHAPASCAN-0. Sa performance dépend de celle d'ATHAPASCAN-0. Pour analyser le gain apporté à ATHAPASCAN-1 par ATHAPASCAN-SMP, nous avons choisi trois applications couramment utilisées par l'équipe ATHAPASCAN-1 : Fibonacci, Jacobi et Gauss. L'intérêt des ces trois applications dans le contexte de cette analyse est leur différent rapport calcul-communication.

[11]Le terme tâche dans le contexte ATHAPASCAN-1 représente une unité de travail à réaliser. Dans le contexte ATHAPASCAN-0, le terme tâche fait référence à un processus lourd. En suivant la terminologie ATHAPASCAN-1, les tâches ATHAPASCAN-0 sont synonymes au terme noeud.

La suite de Fibonacci

La suite de Fibonacci est basée sur l'évaluation de $F(x) = a_{x-1} + a_{x-2}$ pour tout $x \geq 2$ en considérant $F(0) = 0$ et $F(1) = 1$. Le programme ATHAPASCAN-1 pour évaluer cette suite emploie un algorithme récursif. Ainsi lorsque le programme ATHAPASCAN-1 s'exécute, il crée une tâche initiale pour calculer Fibonacci de x. Cette tâche créera deux tâches, une pour évaluer Fibonacci de $x - 1$ et l'autre pour Fibonacci de $x - 2$. Cette procédure est exécutée jusqu'à ce qu'un seuil de découpe soit atteint. A partir de cet instant, les tâches calculent Fibonacci de façon séquentielle. La figure 6.11 illustre cette procédure.

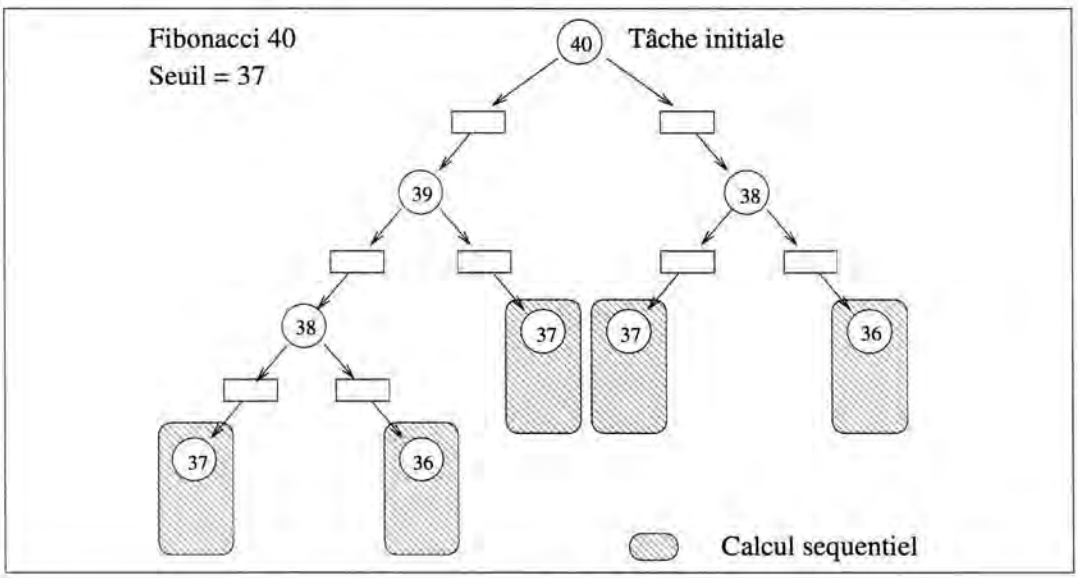

Figure 6.11 *Le graphe de tâches pour Fibonacci*

Nous avons réalisé l'exécution de Fibonacci en utilisant au niveau applicatif un ordonnanceur de type glouton[35]. Dans ce cas, lorsqu'on lance une application ATHAPASCAN-1 avec n noeuds, un de ces noeuds initie la découpe de création de tâches. Les autres $n - 1$ noeuds demandent du travail. Le noeud initial envoie une tâche au demandeur lequel effectue la même découpe. Étant donné ce schéma, cette application communique seulement dans la phase initiale de découpe de tâches : les noeuds oisifs demandent du travail, les noeuds qui réalisent la découpe le distribuent. Quand le seuil est atteint, aucune autre communication n'est faite par une tâche[12]. Fibonacci constitue donc une application où le calcul tend à prédominer sur les communications. Le grain de calcul est réglé par le seuil s de découpe. Dans ce cas, la scrutation réalisée par le démon ATHAPASCAN-0 tend à s'exécuter inutilement. Le tableau 6.9 donne les différents temps d'exécution de Fibonacci sur ATHAPASCAN-0 et ATHAPASCAN-SMP en utilisant différents seuils s. La dernière ligne de ce tableau nous donne la durée d'exécution du code parallèle sur un seul noeud et un seul processeur virtuel c'est à dire T_1 (section 5.3). Pour éliminer toute

[12] Sauf à la fin pour envoyer les résultats, ou demander plus de travail.

interférence du démon de communication ATHAPASCAN-0, nous avons obtenu T_1 en désactivant la scrutation (mode dégénéré - section 5.2.2).

Tableau 6.9 *Temps d'exécution Fibonacci $F(40)$ sur deux noeuds (Biprocesseur, réseau ethernet 100 Mpbs)*

al_pool=1	s=35	s=30	s=25	s=20
ATHAPASCAN-0	6.77±0.10s	6.23±0.07s	6.22±0.06s	6.73±0.05s
ATHAPASCAN-SMP	6.35±0.03s	5.80±0.01s	5.80±0.02s	6.50±0.01s

al_pool=2	s=35	s=30	s=25	s=20
ATHAPASCAN-0	5.17±0.11s	4.53±0.07s	4.52±0.07s	4.91±0.05s
ATHAPASCAN-SMP	4.56±0.03s	4.12±0.03s	4.13±0.03s	4.65±0.03s

al_pool=4	s=35	s=30	s=25	s=20
ATHAPASCAN-0	4.50±0.07s	4.20±0.08s	4.02±0.06s	4.65±0.08s
ATHAPASCAN-SMP	4.21±0.05s	3.70±0.04s	3.76±0.04s	4.35±0.05s

T_1	11.67±0.20s	11.64±0.20s	11.77±0.20s	12.90±0.25s

En analysant les résultats obtenus, on constate que toutes les exécutions ont une meilleure performance pour des découpes intermédiaires. Ceci est expliqué par le fait qu'une découpe avec un seuil $s = 35$ ne génère pas assez de parallélisme, et que pour une découpe plus fine ($s = 20$) le surcoût de la génération du graphe de données devient important, une granularité plus adéquate est obtenue par des découpes intermédiaires. L'effet de la gestion du graphe de données est discuté en [81].

Ensuite, à partir des résultats dans la fourchette optimale, on peut vérifier l'hypothèse du surcoût introduit par le démon de communication. Si l'on considère que l'ordonnanceur applicatif d'ATHAPASCAN-1 partage équitablement T_1 entre les *threads* qui constituent le pool d'exécution, on peut espérer que chaque *thread* s'exécute par une durée T_1/k, où k est le nombre de *threads* du pool. Si nous prenons, par exemple, le cas correspondant à un pool de 4 *threads* nous avons globalement 8 *threads* de calcul (4 sur chaque noeud) donc chaque *thread* s'exécute en 1.45 s (11.64/8). Comme le démon de communication ATHAPASCAN tend à s'exécuter en une même période de temps (cf. section 5.4) nous avons un temps total d'exécution de 10×1.45s (4 *threads* de calcul plus un démon par noeud), lequel est divisé entre 4 processeurs (2 de chaque noeud), ceci nous amène à un temps total d'exécution de $(10 \times 1.45)/4 = 3.62$s contre 3.7 s mesuré.

A l'exemple de l'application Takakaw, où le temps de calcul prédomine sur celui des communications, la performance de deux versions d'ATHAPASCAN-0 sont similaires. La différence, toujours en faveur d'ATHAPASCAN-SMP, peut être expliquée par les temps épargnés dans la phase initiale de communication. Sur certaines découpes, nous avons aussi des messages de demande et acquittement de travail.

Le calcul de Jacobi

Le calcul de Jacobi est une méthode numérique itérative employée pour évaluer le Laplacien d'une fonction. Le Laplacien aide à l'analyse du comportement de plusieurs phénomènes physiques comme par exemple la transmission de chaleur sur des surfaces. Plusieurs algorithmes sont employés pour évaluer Jacobi. Celui implanté en ATHAPASCAN-1, considère au départ un tableau unidimensionel de taille d. Le calcul de Jacobi est fait en i itérations, l'itération i dépend des valeurs de l'itération $i-1$. La figure 6.12 montre schématiquement son fonctionnement.

On exécute le Jacobi ATHAPASCAN-1 sur n noeuds. La tâche initiale (noeud zéro), à la différence du cas précédent de Fibonacci, réalise la création complète du graphe de tâches. Les $n-1$ noeuds restants demandent du travail à la tâche initiale. L'ordonnanceur utilisé est à nouveau de type glouton.

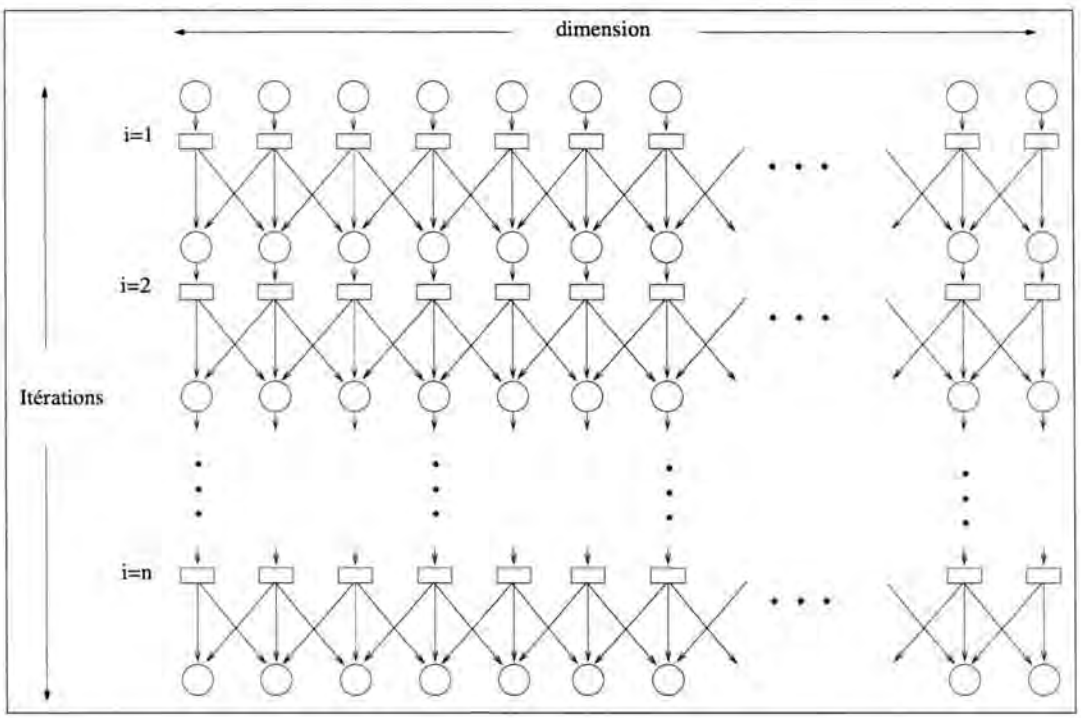

Figure 6.12 *Le graphe de tâches pour Jacobi*

Pour chaque demande de travail, l'ordonnanceur ATHAPASCAN-1 envoie vers le demandeur une seule unité de travail composée par les valeurs des trois arcs rentrants sur un arête du graphe de tâches (fig. 6.12). Le résultat est renvoyé vers la tâche initiale. Par rapport à Fibonacci, Jacobi met en jeu un grand volume de communications pour chaque calcul. La granularité de calcul est inférieure à celle des communications. Ces communications sont caractérisées par de petits messages (typiquement 32 octets). Dans ce cas, la réactivité d'ATHAPASCAN-0 devient importante. Le tableau 6.10 montre les résultats obtenus lors de plusieurs exécutions de Jacobi en variant le nombre d'itérations (i) et la dimension du tableau (d).

Tableau 6.10 *Temps d'exécution Jacobi sur deux noeuds (Biprocesseur, réseau ethernet 100 Mpbs, mode c2c)*

	d=64		d=128	
a1_pool=1	i=10	i=20	i=10	i=20
ATHAPASCAN-0	8.78±0.17s	16.61±0.4s	16.5±0.43s	32.5±0.39s
ATHAPASCAN-SMP	5.54±0.04s	10.7±0.05s	11.3±0.06s	21.54±0.07s

	d=64		d=128	
a1_pool=2	i=10	i=20	i=10	i=20
ATHAPASCAN-0	7.59±0.59s	13.47±0.72s	14.5±1.18s	25.16±0.52s
ATHAPASCAN-SMP	4.62±0.02s	8.89±0.02s	9.10±0.05s	17.67±0.05s

	d=64		d=128	
a1_pool=4	i=10	i=20	i=10	i=20
ATHAPASCAN-0	7.41±0.7s	11.34±0.34s	12.7±0.67s	24.16±1.27s
ATHAPASCAN-SMP	4.65±0.05s	8.96±0.11s	9.55±0.1s	18.7±0.19s

T_1	5.59±0.04s	10.94±0.06s	11.62±0.06s	21.06±0.07s

On constate immédiatement deux choses à partir des valeurs du tableau 6.10. La première est un écart plus important de performance entre les deux version d'ATHAPASCAN. La deuxième est que les temps obtenus par l'exécution parallèle sont du même ordre de grandeur que le temps T_1. Ensuite nous observons aussi que le gain résultant de l'augmentation du nombre de tâches dans le pool d'exécution A1 n'est pas significatif. Nous pouvons expliquer cela par le grand déséquilibre entre communications et calculs de cette application.

Cette implantation de Jacobi est caractérisée par un volume très important de messages. Le noeud initial (zéro) est le seul à avoir accès au graphe de tâches. Le deuxième noeud réalise des demandes de travail, d'autant plus que le nombre de processeurs virtuels augmente (pool d'exécution). La granularité de calcul est très petite. L'ordonnanceur du noeud zéro est saturé par les demandes de travail. C'est seulement à la fin de chaque tâche que, l'ordonnanceur du noeud zéro peut acquitter les demandes de travail de l'autre noeud. L'ordonnanceur ATHAPASCAN-1 depose des requêtes de communication et les acquite en permanence. Il est donc toujours en conflit avec le démon de communication ATHAPASCAN-0 par les files de requêtes. Celui-ci devient le goulot d'étranglement du système.

Cette application présente une très mauvaise découpe calcul-communication. La solution immédiate est de changer l'ordonnanceur applicatif (en effet un ordonnanceur fixe à la place du glouton offre de meilleurs résultats[35]). Toutefois, cette mauvaise découpe fait ressortir le gain du démon ATHAPASCAN-SMP par rapport à la version originale, gain obtenu par la réduction des commutations de contextes due à la compétition sur la liste de requêtes.

Le calcul de Gauss

Gauss est une méthode numérique utilisée pour la résolution de systèmes d'équations linéaires. L'implantation de cette méthode en ATHAPASCAN-1 considère au départ une matrice carrée qui est partagée en sous matrices (blocs) $b \times b$. Le noeud zéro construit le graphe de tâches en associant un ensemble de tâches à un bloc de la matrice. Ensuite l'ordonnanceur applicatif affecte du travail aux noeuds en distribuant les tâches associées aux blocs d'une façon cyclique par colonnes. Par exemple, si l'on suppose une division en 4 noeuds (fig. 6.13), le noeud zéro reçoit les tâches associées au bloc b_0, le noeud 1 celles du bloc b_1, le noeud 2 du bloc b_4, etc.

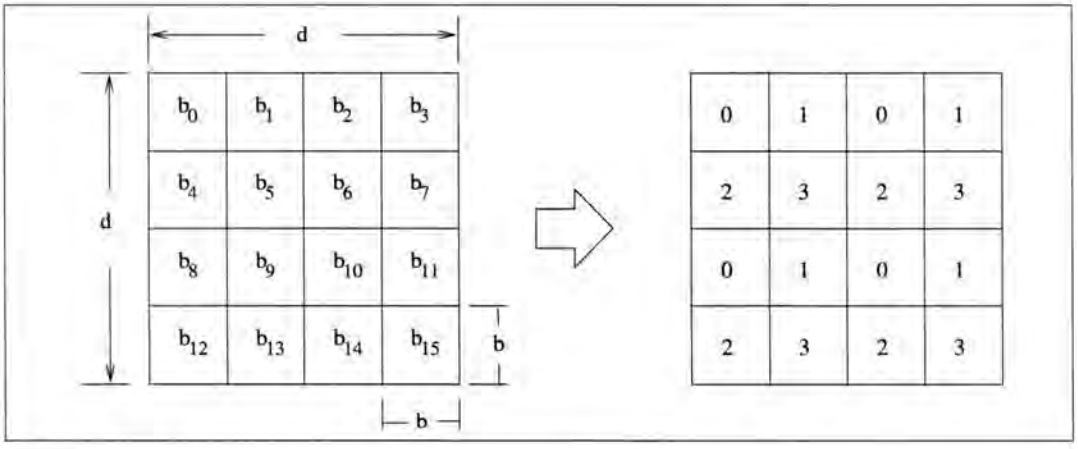

Figure 6.13 *L'ordonnancement* bi_dim *pour l'évaluation de Gauss*

L'algorithme pour évaluer gauss génère un volume de communications $O(d^2)$ et un poids de calcul $O(2/3d^3)$. Le nombre de communications est $O(d^2/b^2)$. En réalité, chaque communication demande un nombre additionel d'échanges de messages pour la gestion de tâches. Par rapport aux deux cas précédents, Gauss offre un rapport calcul-communication plus équitable. Nous avons donc exécuté Gauss sur les deux versions d'ATHAPASCAN, et les résultats obtenus sont présentés au tableau 6.11.

En observant ces valeurs, on vérifie une fois de plus la meilleure performance d'ATHAPASCAN-SMP. L'écart entre les deux versions est plus important à mesure qu'on augmente le nombre de messages par augmentation de la dimension d de la matrice. Ceci confirme que la gestion de communications d'ATHAPASCAN-SMP est plus efficace que celle de la version originale. Le nombre de tâches à calculer sur chaque noeud est approximativement le même, donc on peut supposer un partage équitable de calcul. Globalement nous avons 4 processeurs, 2 démons de communications (un sur chaque noeud), et $k \times 2$ processeurs virtuels (k par noeud). En utilisant l'hypothèse du modèle de coût de la section 5.3 on obtient les estimations suivantes pour le temps d'exécution ($d = 2000, b = 200$) :

- *a1_pool=1 :* k=2, p=4 $T_l = 178.98/2 \approx 88.50s$;

Tableau 6.11 *Temps d'exécution Gauss sur deux noeuds (Biprocesseur, réseau ethernet 100 Mpbs)*

	d=1000		d=2000	
a1_pool=1	b=100	b=200	b=100	b=200
ATHAPASCAN-0	19.88±0.32s	15.87±0.22s	192.14±15.02s	130.70±2.94s
ATHAPASCAN-SMP	15.31±0.01s	14.23±0.01s	118.35±0.20s	102.71±0.25s

	d=1000		d=2000	
a1_pool=2	b=100	b=200	b=100	b=200
ATHAPASCAN-0	16.27±0.53s	14.39±0.47s	178.26±11.44s	109.01±6.5s
ATHAPASCAN-SMP	12.09±0.13s	11.36±0.15s	91.08±1s	78.05±0.35s

	d=1000		d=2000	
a1_pool=4	b=100	b=200	b=100	b=200
ATHAPASCAN-0	13.97±0.4s	13.48±0.2s	160.10±6.3s	90.70±2.86s
ATHAPASCAN-SMP	11.79±0.12s	11.67±0.17s	89.91±0.8s	72.21±0.35s

T_1	21.64±0.01s	22.29±0.01s	177.18±0.56s	176.98±0.4s

- *a1_pool=2 :* k=4, p=4 $T_l = [(k+2)/(p \times k)]T_1 \approx 67$;
- *a1_pool=4 :* k=8, p=4 $T_l = [(k+2)/(p \times k)]T_1 \approx 56$;

On constante donc que toutes les valeurs estimées sont inférieures à celles mesurées. Cette différence provient du fait que les *tâches* se synchronisent de temps en temps par des communications et notre modèle de coût du démon ne rend pas compte de ces synchronisations.

6.5 Bilan

Nous avons procédé à l'évaluation d'ATHAPASCAN-SMP et nous avons pu vérifier qu'il est plus efficace que la version version originale sur des machines multiprocesseurs. Les résultats obtenus nous permettent de dire :

- Les méthodes d'intégration de la multiprogrammation légère et des communications développées sur des machines monoprocesseurs sont toujours valables dans un cadre multiprocesseur. Néanmoins il faut prendre en compte les aspects propres à la programmation sur ce type de machines, comme l'exécution simultanée des *threads*, les accès à des ressources partagées et leur influence sur la contention et la compétition sur des verrous, etc. La prise en compte de ces aspects sont la raison des améliorations que nous avons apportées à ATHAPASCAN-0.
- Un problème de réglage fin s'impose sur les multiprocesseurs. Nous avons montré le gain de performance dû à la réduction de commutations de contexte

inutiles par l'emploi de la technique de l'attente active (*spin-lock*). Toutefois trouver la durée idéale de l'attente active est assez délicat et est fortement dépendant de la machine cible[13]. Un mauvais choix peut provoquer une dégradation.

- Le parallélisme réel disponible sur une machine de type multiprocesseur permet un recouvrement aisé des communications par des calculs en exploitant la multiprogrammation légère.
- Pour la scrutation les techniques restent les mêmes indépendemment du type de la machine. Les problèmes d'efficacité et réactivité correspondant ne sont pas résolus.

Un des principaux objectifs de l'environnement de programmation ATHAPASCAN est celui de la portabilité. Pour atteindre cet objectif, nous avons basé notre implantation sur les noyaux MPI et POSIX *threads*, disponibles sur un grand nombre de plate-formes. Ce choix s'est montré « gagnant » dans le sens où ATHAPASCAN est aisement porté d'une plate-forme à l'autre. Toutefois cette portabilité a un prix : les noyaux MPI et POSIX *threads* sont vus comme des « boîtes noires » et nous n'avons aucun contrôle direct sur leur fonctionnement. Par exemple, MPI ne remonte pas un signal indiquant la réception d'un message.

Le point le plus gênant en présence d'un noyau de *threads* 1 :1 (*threads* systèmes) est que nous n'avons aucun contrôle sur la façon dont les *threads* sont ordonnancés. Ceci a une conséquence immédiate sur la réactivité et sur l'efficacité de la procédure de scrutation. La source d'inefficacité est le grand nombre d'exécutions inutiles de la scrutation. L'ordonnancement des *threads* fixe la réactivité.

Le problème de la réactivité restera ouvert tant qu'on est incapable d'influencer ou contrôler l'ordonnanceur. De façon transparente à l'utilisateur on est simplment capable de garantir un avancement des communications et de donner les conseils suivants : (a) un nombre petit de *threads* (2-4) implique un surcoût plus important de la scrutation ; (b) un nombre plus important de *threads* nuit à la réactivité. Le programmeur a donc un compromis efficacité/réactivité à gérer. Ce compromis se traduit aussi sur le choix d'une découpe adéquate des calculs et communications. Trouver le bon compromis reste malhereusement une charge laissée au programmeur de l'application parallèle.

Nous avons proposé un modèle de coût de la scrutation (chapitre 5) qui est une bonne approximation si peu de synchronisations existent entre *threads*. Dans le cas contraire, les prédictions de coût sont incorrectes. En effet, un tel comportement est très difficile à modéliser parce qu'il est fortement dépendant de l'application. Néanmoins le modèle proposé s'adapte bien aux applications dont la synchronisation n'est pas forte et s'avère utile pour une meilleur compréhènsion du comportement global de l'application.

[13] D'ailleurs un simple changement de fréquence d'horloge suffit pour dérégler.

7

Conclusion et perspectives

"En toute chose il faut considérer la fin"
Jean de La Fontaine, Le renard et le bouc

L'environnement de programmation parallèle ATHAPASCAN, développé dans le cadre du projet APACHE, a pour objectif d'offrir un modèle de programmation parallèle pour le calcul haute performance permettant d'obtenir des programmes efficaces et portables. Cet environnement est basé sur un noyau exécutif qui réalise l'intégration des processus légers (*threads*) et des communications : ATHAPASCAN-0. Le présent travail porte sur l'analyse et l'évaluation de ce noyau sur les noeuds multiprocesseurs d'une grappe. Ce dernier chapitre résume les principaux points étudiés et les contributions. Il se conclut en présentant les perspectives et travaux futurs envisageables.

7.1 Bilan général

Ce travail de thèse s'insère dans le contexte de la conception et de la réalisation d'environnements de programmation parallèle distribués pour des grappes de multiprocesseurs (SMP).Un modèle de programmation efficace doit contrôler, ou permettre de contrôler, les deux niveaux de parallélisme offerts par une grappe. Le parallélisme au sein d'un SMP et le parallélisme entre SMPs. Le coût attaché à ce second type est celui des communications nécessaires à sa mise en oeuvre (création de *thread* à distance, échange de données, etc).

La multiprogrammation légère s'avère être le véhicule approprié pour l'exploitation du parallélisme offert par les grappes. Elle permet un recouvrement des

délais de communication par des calculs, et l'exploitation du parallélisme réel offert grâce à l'existence de plusieurs processeurs physiques sur un noeud.

L'état de l'art. Nous avons rappelé en quoi un noyau de *threads* 1 :1 était intéressant pour exploiter le parallélisme SMP. En dépit de l'absence de contrôle sur la stratégie d'ordonnancement des *threads* (sections 4.2.3, 5.2.2,6.3.1), nous avons montré qu'il était possible (sur notre plate-forme) de « prédire » dans une certaine mesure le temps processeur alloué à chaque *thread* d'un programme.

Nous avons rappelé les fonctions et caractéristiques de fonctionnement d'un noyau de communication tel que MPI. Nous avons montré que des « jeux de test » permettaient de caractériser l'efficacité de la communication parfaitement.

Nous avons alors posé le probléme de l'intégration de la communication à un noyau de *threads*. C'est à dire le problème de l'utilisation de la multiprogrammation légère pour recouvrir par du calcul les délais de transfert. Nous avons fait le point (chapitre 4) sur les différentes techniques d'intégration et leur point critique : la réactivité à la disponibilité du réseau (émission) ou à l'arrivée d'un message (réception). D'une manière générale toutes les méthodes (message actif, *upcall*, *popup*) butent sur ce problème. Nous avons alors décrit comment certain projets dont le nôtre (ATHAPASCAN-0) tentent de résoudre ce problème : « invoquée de façon appropriée » assure la progression des communications et la synchronisation des *threads* communicants. Tous les projets proposent alors différentes façons « d'invoquer de façon appropriée » cette fonction en justifiant leurs choix par des expériences parfois contraditoires (cf. section 4.3).

Le bilan est clair. Une bonne intégration peut être atteinte :

- soit en fournissant une intéraction entre l'interface réseau et l'application parallèle. Ceci suppose une évolution des noyaux de système et de *threads* actuels. Des projets comme Panda (section 4.3.3) s'y attachent.
- soit en trouvant une solution efficace au niveau applicatif au problème de « tester et faire avancer ».

Contribution de la thèse. Notre travail de thèse porte sur le développement d'ATHAPASCAN-0 pour une grappe où les noeuds sont des SMP. Ceci a requis la réalisation d'une nouvelle version d'ATHAPASCAN-0 tenant compte des spécificités d'un SMP et en particulier les conflits d'accès parallèles aux verrous et variables partagées. Nous avons montré comment une diminution des conflits par une diversification des verrous et l'usage de *spin lock* amélioraient le fonctionnement d'ATHAPASCAN-0, tant sur des jeux de test que nous avons développés (section 5.2) que sur des applications du projet APACHE (section 6.4).

Outre cet objectif d'avoir une version ATHAPASCAN-SMP, nous avions deux autres objectifs :

- concevoir une fonction de « tester et faire avancer » générale et efficace ;

– caractériser dans quelle mesure le parallélisme d'un SMP permettait d'améliorer non seulement le calcul mais aussi la communication.

En ce qui concerne la définition d'une fonction de scrutation efficace et générale le résultat est assez négatif. Nous avons proposé une méthode de « tester et faire avancer » des communications assurée par un *thread* démon **tournant en permanence**. Nous avons montré que cette méthode présentait un surcoût chiffrable et dépendant du nombre de *threads* de calcul (section 5.3) et du travail de calcul à faire (et en particulier du chemin critique de calcul). Ce résultat négatif est particulièrement visible sur un monoprocesseur. Par contre, son effet est atténué sur un multiprocesseur où le démon peut tourner en parallèle du calcul.

De façon pragmatique, nous avons fourni au programmeur la possibilité de retarder le démon de communication en le faisant tourner à échéance (fixée par le programmeur) ou encore de le stopper. Dans ce cas, l'application doit assurer la « tester et faire avancer » de façon appropriée (via la procédure *a0advance()*).

Malgré l'échec de la définition et de la mise en oeuvre d'une solution efficace pour la fonction « tester et faire avancer » au niveau applicatif, les tests (section 5.2) et applications (section 6.4) ont montré qu'il était « facile » d'exploiter le parallélisme SMP et les possibilités de recouvrement calcul-communication. Ainsi, pour battre un programme simple écrit en ATHAPASCAN-0, il fallait produire un programme MPI bien plus complexe. Ceci nécessitait plusieurs itérations d'essai/erreur avant d'obtenir un gain sur le programme ATHAPASCAN-0.

Ces résultats confirment bien que l'intégration *thread* plus communications est une solution à la programmation parallèle de grappes de SMP.

7.2 Travaux futurs & perspectives

Il dérive assez directement de ce travail que l'échec dans la conception d'une fonction de scrutation a deux causes :

– l'impossibilité de bloquer le *thread* de scrutation jusqu'à une transition d'état de la communication caractérisant un travail à effectuer. Cette impossibilité engendre les surcoûts d'exécution.

– l'impossibilité de garantir des échéances précises de traitement des transferts. La cause en est l'absence de contrôle de l'ordonnancement[1] des *threads* chargés de ces transferts. Cette impossibilité implique des surcoûst d'amorçage et des débits atteints qui sont inférieurs aux débits offerts par le réseau.

La correction de ces défauts nécessite soit d'intervenir au niveau système en intégrant les bibliothèques (MPI) au système soit de revoir l'intéraction système application (signalisation des événement de communication) et le contrôle des *threads*

[1] sur les noyaux 1 :1 actuels

(priorité). Cette intégration doit aussi se faire en coordination avec les projets développant de nouvelles interfaces de communications dont le but est de supprimer les surcoûts actuels des protocoles de communication.

La définition de tels protocoles n'est pas triviale car il faut traiter plusieurs points : (a) le choix de la méthode de transfert de/vers le réseau de données le plus approprié (DMA versus réseau mappé en mémoire) ; (b) le choix de mécanismes de translation d'adresses pour transmettre des blocs non contigus de données évitant les copies dans un tampon ; (c) la définition de mécanismes de protection, principalement dans les environnements multiprogrammés, évitant la « lourdeur » d'appels systèmes au niveau applicatif ; (d) méthode pour la signalisation des événements réseau ; et (e) fiabilité pour l'envoi et réception des messages. Plusieurs projets de recherche ont été ménés dans cette voie ces dernières années, par exemple : AM-II[44], FM[126], U- NET[165], PM[154], VMMC[59], LFC[18], Trapeze[7], et BIP[132].

La diversité des solutions proposées par ces projets, sans qu'un consensus soit dégagé, a poussé les constructeurs Intel, Microsoft et Compaq à définir un standard pour l'interconnexion de grappes : VIA (*Virtual Interface Architecture*)[62].

Une grappe de SMP, construite à partir d'ordinateurs (PC) et d'un réseau haut débit standard du commerce (Ethernet, myrinet, SCI, etc) , est donc un moyen pour obtenir une machine parallèle de puissance comparable aux machines parallèles spécialisées (Cray T3E, SGI, etc). Pour cela, il faut assembler en grappe plusieurs centaines de PCs. Les équipes APACHE[2], ReMaP[3], et SIRAC[4]ont un projet commum de déploiment d'une grappe de 200 PCs.

En plus des problèmes techniques de déploiement d'une telle grappe, il nous faut savoir comment un environnement tel que ATHAPASCAN-0 « résiste » au portage à l'echelle. Un problème de vérification de ce passage à l'échelle sera le développment de techniques de mesures et le développment de programmes de tests appropriés. Notre expérience modeste dans le cadre de cette thèse nous incite à considérer que ceci risque d'être un travail difficile et délicat.

[2]http ://www.inria.fr/Equipes/APACHE-fra.html
[3]http ://www.inria.fr/Equipes/REMAP-fra.html
[4]http ://www.inria.fr/Equipes/SIRAC-fra.html

Annexe A

Mesures & interprétation

L'estimation de moyennes et l'intervalle de confiance : Une moyenne $\overline{x}_n$ obtenue à partir de n mesures d'un phénomène sur un système fournit une estimation de la moyenne « réelle » du comportement de ce système vis à vis de ce phénomène. Ceci, en termes statistiques, signifie que la moyenne $\overline{x}_n$ obtenue sur un échantillon de la population ne représente qu'une estimation de la moyenne μ de cette population (inconnue au préalable). En réalisant k observations, on obtient k estimations différentes. La question qui se pose est de déterminer dans quelle mesure elles approchent la moyenne μ de la population.

Il est impossible de déterminer cela de façon exacte à partir d'un nombre fini d'échantillons. On va donc plutôt chercher à encadrer la valeur cherchée. Pour ceci, on va essayer de déterminer un intervalle d'erreur I autour de la moyenne estimée $\overline{x}_n$ qui nous garantit avec une probabilité α que la moyenne réelle μ est comprise dans cet intervalle. En d'autres termes, α est le degré de confiance de l'intervalle I c'est à dire que la probabilité $P(\mu \in I)$, que l'intervalle d'erreur I contienne la moyenne μ de la population, est supérieure ou égale à α. Pour un échantillon « assez grand » ($n > 100$), l'intervalle de confiance se calcule par :

$$I = \left]\overline{x}_n - \varepsilon_\alpha \frac{\tilde{\sigma}_n}{\sqrt{n}}, \overline{x}_n + \varepsilon_\alpha \frac{\tilde{\sigma}_n}{\sqrt{n}}\right[\tag{A.1}$$

où, $\overline{x}_n$ est l'estimateur de la moyenne μ de la population calculé par :

$$\overline{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i, \tag{A.2}$$

et $\tilde{\sigma}_n$ est l'estimateur de l'écart type calculé par :

$$\hat{\sigma}_n = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x}_n)^2}, \tag{A.3}$$

et ε_α dépend du niveau de confiance α souhaité. Les valeurs de ε_α en fonction de α sont des données ou se calculent par la formule :

$$\alpha = \int_{-\varepsilon_\alpha}^{\varepsilon_\alpha} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx.$$

Ce principe sert donc à calculer à partir d'une série de mesures d'un système une moyenne approchant la valeur cherchée avec un dégré de confiance fixé a priori.

Interprétation des mesures : Il devient alors possible de comparer deux systèmes, A et B, dans des conditions similaires. Trois situations sont possibles en fonction des intervalles de confiance trouvés :

- Ils ne se recoupent pas : on peut affirmer qu'un système **est supérieur** à l'autre avec un degré de certitude donné par le produit $\alpha_A \times \alpha_B$ des niveaux de confiance utilisés pour obtenir les mesures sur les deux systèmes ;
- Ils se recoupent ET la moyenne de l'un d'eux est comprise dans l'intervalle de confiance de l'autre : on peut conclure que les deux systèmes **sont équivalents** ;
- Ils se recoupent MAIS la moyenne d'un système n'est pas comprise dans l'intervalle de confiance de l'autre : aucune conclusion ne peut être faite a priori. Il faut suivre une procédure appelée **t-test** [101] pour les distinguer. Dans ce travail nous n'appliquerons pas ce principe, et nous considérerons les deux systèmes comme équivalents[1] (cette situation s'est produite très rarement sur les tests effectués).

Détermination de la taille d'échantillon : Le niveau de confiance de ces conclusions dépend de la qualité des estimateurs de base ($\overline{x}_n, \hat{\sigma}_n^2, \hat{\sigma}$), c'est à dire de la taille de l'échantillon analysé. Plus grand est l'échantillon, plus élevée est la confiance. Toutefois, analyser de grands échantillons signifie plus de ressources utilisées (efforts, heures CPU, etc). Il faut donc déterminer un nombre suffisant d'éléments à analyser de façon à ce que le niveau de confiance souhaité soit atteint.

L'estimation de μ par $\overline{x}_n$ avec une précision relative de $\pm r\%$ signifie que l'intervalle de confiance doit être compris entre :

[1] Mes sincères excuses aux statisticiens

$$I = \left]\overline{x}_n \left(1 - \frac{r}{100}\right), \overline{x}_n \left(1 + \frac{r}{100}\right)\right[$$

La taille de l'échantillon n, nécessaire pour satisfaire cette condition est calculée par :

$$\overline{x}_n \pm \varepsilon_\alpha \frac{\hat{\sigma}_n}{\sqrt{n}} = \overline{x}_n \left(1 \pm \frac{r}{100}\right)$$

$$\varepsilon_\alpha \frac{\hat{\sigma}_n}{\sqrt{n}} = \overline{x}_n \frac{r}{100}$$

$$n \geq \left(\frac{100\, \varepsilon_\alpha\, \hat{\sigma}_n}{\overline{x}_n\, r}\right)^2 \qquad \text{(A.4)}$$

Toutes les mesures présentées dans cette thèse ont été faites selon cette méthodologie. Le niveau de confiance α adopté est de 90% ($\varepsilon_\alpha = 1.645$). Les estimateurs de base $\overline{x}_n$, $\hat{\sigma}_n^2$, et $\hat{\sigma}$ ont été obtenus à partir d'un échantillon initial de taille 100 (toujours en respectant la relation donnée par l'équation A.4).

Annexe B

La plate-forme d'expérimentation

La figure B.1 montre de façon schématique les machines disponibles au Laboratoire de Modélisation et Calcul (LMC) sur lesquelles nous avons réalisé les expériences présentées dans cette thèse. Notre plate-forme se compose de :

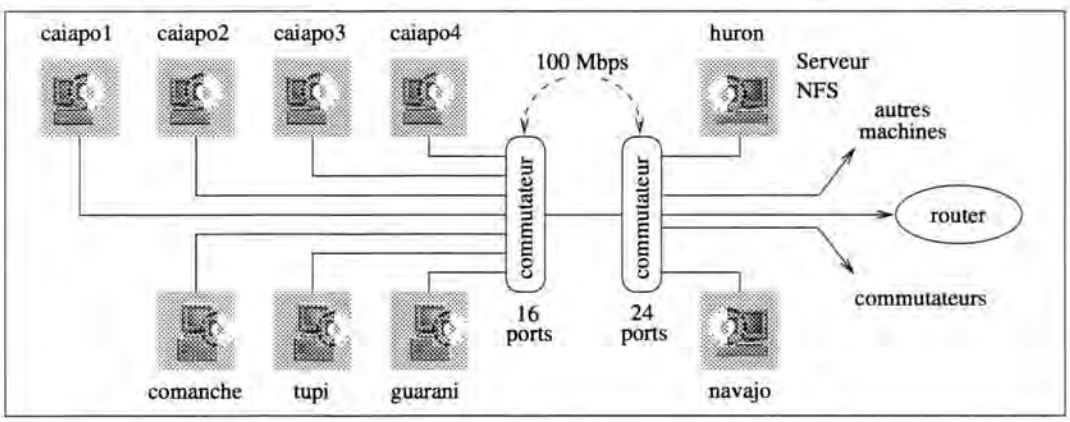

Figure B.1 *La plate-forme d'expérimentation du LMC*

- 4 machines monoprocesseurs à base de Pentium à 133 Mhz, 64 Moctets de mémoire vive, 256k octets mémoire cache. Le système d'exploitation utilisé est SunOs 5.6 (solaris 2.6). Ces machines sont appelées *caiapo1*,*caiapo2*,*caiapo3* et *caiapo4* ;
- 2 machines biprocesseurs à base de PentiumII à 333 Mhz, 128 Moctets de mémoire vive, 512k octets mémoire cache. Le système d'exploitation utilisé est SunOs 5.6 (solaris 2.6). Ces machines sont appelées *guarani* et *tupi* ;
- 1 machine quadriprocesseur à base de pentiumPro à 200 Mhz, 256 Moctets de mémoire vive,512k octets mémoire cache. Le système d'exploitation utilisé est SunOs 5.6 (solaris 2.6). Cette machine est appelée *comanche* ;
- 1 biprocesseur SuperSparc à 60 Mhz, 224 Moctets de mémoire vive, 1 Moctets de mémoire cache. Le système d'exploitation utilisé est SunOs 5.6 (solaris 2.6). Cette machine est appelée *hopi* ;

- 1 biprocesseur UltraSparc-II à 295 Mhz, 512 Moctets de mémoire vive, 4 Moctets mémoire cache. Le système d'exploitation utilisé est SunOs 5.6 (solaris 2.6). Cette machine est appelée *huron* ;

Toutes les machines sont reliées à un réseau Fast Ethernet (100 Mbps) des commutateurs. Les machines Intel sont toutes sur un même commutateur 16 ports (Elles sont les seules machines sur ce commutateur). Ce commutateur est relié à celui auquel les machines Sparc (*huron* et *navajo*) sont connectées. La machine *huron* est le serveur NFS de cet ensemble.

La bibliothèque de communication utilisée est LAM MPI version 6.3. Le noyau POSIX *threads* de Solaris 2.6 est le noyau de processus légers utilisé.

Annexe C

Le calcul de la fractale de Mandelbrot

Le calcul de la fractale de Mandelbrot se parallélise facilement. En effet, l'appartenance à l'ensemble d'un point C sur un plan imaginaire $(x, i.y)$ est indépendant des points voisins. Soit $C = x + i.y$ un point du plan, la question est : est-ce que la suite $Z_{n+1} = Z_n^2 + C$ converge (avec $Z_0 = 0$) ? Si un Z_i possède un module supérieur à 2 alors la suite va diverger, et on colorie le point C avec la couleur i. Si, au bout d'un nombre donné (suffisamment grand) les modules des Z_i sont toujours inférieurs à 2, alors on considère que la suite ne diverge pas, et on colorie le point C en noir (C appartient alors à la fractale de Mandelbrot). On peut donc imaginer une découpe du calcul à grain très fin, c'est à dire qu'il peut y avoir autant d'« unités de calcul » que de points considérés. Cependant il est plus approprié de découper le plan considéré en régions regroupant des points de façon à accroître le grain de calcul.

Nous pouvons implanter les « unités de calcul » selon les paradigmes de programmation suivants :

- **Par partage de mémoire :** la méthode de parallélisation est un système à pile de travail (*workpile*), c'est à dire un certain nombre de *threads* accèdent à une structure centralisée qui détient le travail à exécuter. Un *thread* récupère un travail (ici une région), calcule, et ensuite cherche sur cette pile une nouvelle région à calculer. Cette procédure est répétée jusqu'à qu'il n'y ait plus de travail à réaliser.
- **Par échange de messages :** la méthode de parallélisation est la ferme de processus où un processus maître distribue du travail, c'est à dire des régions à des processus esclaves. Les esclaves sont bloqués sur une opération de *receive* en attendant l'envoi d'une région de la part du maître . Lorsqu'ils reçoivent une région, ils calculent et à la fin renvoient le résultat vers le maître. Tant qu'il reste des régions à calculer, le maître en envoie aux esclaves.

Pour réaliser nos mesures nous avons effectué différentes variantes du calcul de

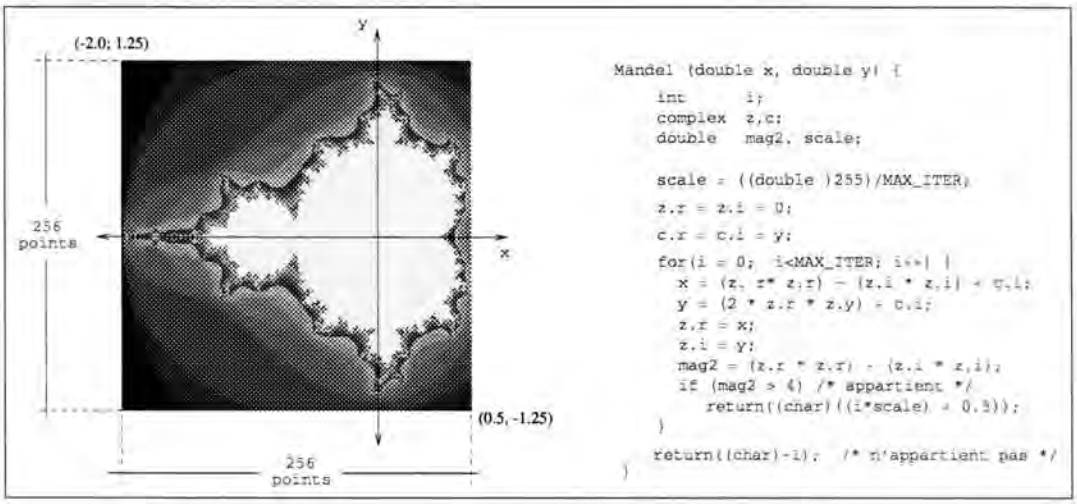

Figure C.1 *La fractale de Mandelbrot*

Mandelbrot selon la caractéristique à analyser. Ces différentes implantations seront détaillées au moment de leur emploi. Ces variantes ont en commun le fait que nous utilisons 25 itérations pour déterminer l'appartenance d'un point à l'ensemble de Mandelbrot sur le plan délimité par les points $(-2.0; 1.25i), (0.5; -1.25i)$. Nous utilisons un plan de 256 par 256 points[1]. Cette image est à son tour divisée en régions de taille log_2. La procédure de calcul d'un point à l'ensemble est donnée par la figure C.1.

Mandelbrot synthétique : L'inconvénient de Mandelbrot est que les régions présentent un poids de calcul différent en fonction de l'appartenance ou pas des points à son intérieur à l'ensemble. Pour « uniformiser » le temps de calcul nous avons modifié l'algorithme de la figure C.1 pour supprimer la sortie anticipée de la boucle. Ainsi tous les points, de toutes les régions, provoqueront l'exécution de la boucle *for* dans son intégralité donc un poids de calcul constant. Nous appelons ceci **Mandelbrot synthétique** ou **régulier**.

Nous pouvons facilement modifier le rapport calcul-communication en altérant la taille de la région. Nous avons ainsi le contrôle du « poids » de calcul par la valeur du nombre d'itérations MAX_ITER (fig. C.1) défini à l'exécution par la ligne de commande.

[1]En cas des modifications de ces valeurs nous les expliciterons dans le texte.

Bibliographie

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine : Architecture and performance. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA'95)*, pages 2–13, June 1995.

[2] Gul Agha. *ACTORS : A model of Concurrent computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1990.

[3] Gul A. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems*. PhD thesis, University of Michigan, Computer and Communication Science, 1985. also MIT AI Laboratory Technical Report 844.

[4] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy. Matching Language and Hardware for Parallel Computation in the Linda Machine. *IEEE Transactions on Computers*, 37(8) :921–929, August 1988.

[5] Alliance. Gigabit ethernet overview – A whitepaper. Technical report, Alliance, May 1997.

[6] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, pages 1–6, 1990.

[7] Darrell C. Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, Kenneth G. Yocum, and Michael J. Feeley. Cheating the I/O bottleneck : Network storage with Trapeze/Myrinet. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 143–154, Berkeley, USA, June 15–19 1998. USENIX Association.

[8] T. Anderson. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1) :6–16, January 1990.

[9] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations : effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principle*, pages 95–109. ACM SIGOPS, October 1991. Published in ACM Operating Systems Review Vol.25, No.5, 1991. Also published ACM Transactions on Computer Systems Vol.10 No.1, pp.53–70, Feb. 1992.

[10] F. Andre, M. Le Fur, Y. Maheo, and J.-L. Pazat. The Pandore data-parallel compiler and its portable runtime. *Lecture Notes in Computer Science*, 919 :176–??, 1995.

[11] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. In Akkihebbal L. Ananda and Balasubramaniam Srinivasan, editors, *Distributed Computing Systems : Concepts and Structures*, pages 338–369. IEEE Computer Society Press, Los Alamos, CA, 1992.

[12] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca : A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3) :190–205, March 1992.

[13] Vasanth Bala, Shlomo Kipnis, Larry Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. In Linda R. Petzold Richard F. Sincovec, David E. Keyes, Michael R. Leuze and Daniel A. Reed, editors, *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 862–872, Norfolk, VI, March 1993. SIAM Press.

[14] Daniel J. Berg. Java threads – A whitepaper. Technical report, Sun Microsystems, March 1996.

[15] P.-E. Bernard. *Parallélisation et multiprogrammation pour une application irrégulière de dynamique moléculaire opérationnelle*. Thèse de doctorat en mathématiques appliquées, Institut National Polytechnique de Grenoble, France, October 1997.

[16] G. Berry and G. Gonthier. The ESTEREL synchronous programming language : design, semantics, implementation. Technical Report RR-0842, Inria, Institut National de Recherche en Informatique et en Automatique, 1988.

[17] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and Distributed Computation*, chapter 5, 6.5 and 6.6. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[18] R. Bhoedjang, T. Rühl, and H. Bal. Efficient multicast on myrinet using link-level flow control. In *Proceedings of the 1998 International Conference on Parallel Processing (ICPP '98)*, pages 381–391, Washington - Brussels - Tokyo, August 1998. IEEE USA.

[19] Raoul Bhoedjang and Koen Langendoen. User-space solutions to thread switching overhead. In *First Annual Conference of the Advanced School for Computing and Imaging, Heijen, Netherlands*, pages 397–406, June 1995.

[20] Raoul Bhoedjang, Tim Ruhl, Rutger Hofman, Koen Langendoen, Henri Bal, and Frans Kaashoek. Panda : A portable platform to support parallel programming languages. In USENIX Association, editor, *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV) : September 22–23, 1993, San Diego, California, USA*, pages 213–226, Berkeley, CA, USA, September 1993. USENIX.

[21] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1) :39–59, February 1984.

[22] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *Computer*, 23(5) :35–43, May 1990.

[23] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. C. E. Zhou, K. H. Randall, and Y. Zhou. Cilk : an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8) :207–216, August 1995.

[24] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet : A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1) :29–36, February 1995.

[25] S. Bolis, E. G. Economou, D. Mouzakis, and G. Philokyprou. SBP-net : an integrated voice/data token ring LAN. *ACM Computer Communications*, 16, 8 :494–500, 1993.

[26] Luc Bouge, Jean-Francois Mehaut, and Raymond Namyst. MADELEINE : An efficient and portable communication interface for RPC-based multithreaded environments. Technical Report RR-3459, Inria, Institut National de Recherche en Informatique et en Automatique, 1999.

[27] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John D. Kubiatowicz. Remote queues : Exposing message queues for optimization and atomicity. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 42–53, Santa Barbara, California, July 17–19, 1995. ACM SIGACT/SIGARCH and EATCS.

[28] J. Briat, I. Ginzburg, and M. Pasin. ATHAPASCAN-0B : un noyau exécutif parallèle. *Lettre du Calculateur Parallèle*, 10(3) :273–293, 1998.

[29] Per Brinch Hansen. Distributed processes : A concurrent programming concept. *Communications of the ACM*, 21(11) :934–941, November 1978.

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

[31] R. Calkin, R. Hempel, H.-C. Hoppe, and P. Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4) :615–632, April 1994.

[32] A. Carissimi and M. Pasin. Athapascan : An experience on mixing MPI communications and threads. *Lecture Notes in Computer Science*, 1497 :137–145, 1998.

[33] Gerson G. H. Cavalheiro, Yves Denneuli, and Jean-Louis. Roch. A general modular specification for distributed schedulers. In *EuroPar'98*, Southampton, England., Sept 1998.

[34] Gerson G. H. Cavalheiro, François Galilée, and Jean-Louis Roch. Athapascan-1 : Parallel Programming with Asynchronous Tasks. In *Proceedings of the Yale Multithreaded Programming Workshop*, Yale, USA, june 1998.

[35] Gerson G.H. Cavalheiro. *ATHAPASCAN-1 : Interface générique pour l'ordonnancement dans un environnement*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, Nov 1999.

[36] K. M. Chandy and C. Kesselman. CC++ : A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[37] K. Mani Chandy and C. Kesselman. Compositional C++ : Compositional parallel programming. *Lecture Notes in Computer Science*, 757 :124–132, 1993.

[38] A. S. Charão, I. Charpentier, and B. Plateau. A framework for parallel multithreaded implementation of domain decomposition methods. In *Proceedings of Parallel Computing'99 (to be published)*, Delft, The Netherlands, August 1999.

[39] A. S. Charão, I. Charpentier, and B. Plateau. Programmation par objet et utilisation de processus légers pour les méthodes de décomposition de domaine. *Technique et Science Informatiques* (à apparaître), 1999.

[40] A. S. Charão, I. Charpentier, and B. Plateau. Un environnement modulaire pour l'exploitation des processus légers dans les méthodes de décomposition de domaine. In *11ème Rencontres francophones du parallélisme, des architectures et des systèmes*, Rennes, France, June 1999.

[41] J. Chassin de Kergommeaux, B. de Oliveira Stein, and P. Waille. Mise au point d'applications parallèles irrégulières. In D. Barth, J. Chassin de Kergommeaux, J.-L. Roch, and J. Roman, editors, *ICaRE'97 Conception et mise en œuvre d'applications parallèles irrégulières de grande taille*, chapter 13, pages 267–282. CNRS, December 1997.

[42] M. Christaller, J. Briat, and M. Rivière. Athapascan-0 : concepts structurants simples pour une programmation parallèle efficace. *Calculateurs Parallèles*, 7(2) :173–196, 1995.

[43] Michel Christaller. *Vers un support d'exécution portable pour applications parallèles irrégulières :* ATHAPASCAN-0. PhD thesis, Université Joseph Fourier - Grenoble I, France, Novembre 1996.

[44] Brent N. Chun, Alan M. Mainwaring, and David E. Culler. Virtual network transport protocols for myrinet. Technical Report CSD-98-988, University of California, Berkeley, January 29, 1998.

[45] Keith Clark and Steve Gregory. Parlog : parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1) :1–49, January 1986.

[46] PARKBENCH Committee, Report-1, assembled by Roger Hockney (chairman), and Michael Berry (secretary). Public international benchmarks for parallel computers. Technical Report UT-CS-93-213, Department of Computer Science, University of Tennessee, November 1993.

[47] R. C. B. Cooper and K. G. Hamilton. Preserving abstraction in concurrent programming. *IEEE Trans. on Softw. Eng.*, 14(2) :258, February 1988.

[48] C. Cruz-Neira, J. Leigh, M. Papka, C. Barnes, S. M. Cohen, S. Das, R. Engelmann, R. Hudson, T. Roy, L. Siegel, C. Vasilakis, T. A. DeFanti, and D. J. Sandin. Scientists in wonderland : A report on visualization applications in the CAVE virtual reality environment. In *Proceedings of the Symposium on Research Frontiers in Virtual Reality*, pages 59–66, San Jose, CA, USA, October 1993. IEEE Computer Society Press.

[49] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In IEEE, editor, *Proceedings, Supercomputing '93 : Portland, Oregon, November 15–19, 1993*, pages 262–273, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.

[50] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten Von Eicken. TAM – A compiler controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18(3) :347–370, July 1993.

[51] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP : towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7) :1–12, July 1993.

[52] B. de Oliveira Stein and J. Chassin de Kergommeaux. Environnement de visualisation de programmes parallèles basés sur les fils d'exécution. In D. Trystram, editor, *Actes des 9ièmes Rencontres Francophones du Parallélisme, RenPar9*, Lausanne, May 1997.

[53] B. de Oliveira Stein and J. Chassin de Kergommeaux. Interactive visualisation environment of multi-threaded parallel programs. In *Parallel Computing : Fundamentals, Applications and New Directions*, pages 311–318. Elsevier, 1998.

[54] Thomas A. DeFanti, Ian Foster, Michael E. Papka, Rick Stevens, and Tim Kuhfuss. Overview of the I-WAY : Wide-area visual supercomputing. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(2/3) :123–131, Summer/Fall 1996.

[55] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principle*, pages 122–136. Association for Computing Machinery SIGOPS, October 1991.

[56] B. Dreier, M. Zahn, and T. Ungerer. Rthreads—A uniform interface for parallel and distributed programming. In *Proc. of the 2nd Int'l Conference on Massively Parallel Computing Systems (MPCS'96)*, pages 530–534, May 1996.

[57] B. Dreier, M. Zahn, and T. Ungerer. Parallel and distributed programming with pthreads and rthreads. In *Proc. of the 3rd Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'98)*, pages 34–40, March 1998.

[58] B. Dreier, M. Zahn, and T. Ungerer. The rthreads distributed shared memory system. In *Proc. of the 3rd Int'l Conference on Massively Parallel Computing Systems (MPCS'98)*, April 1998.

[59] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of virtual memory-mapped communication on Myrinet. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS-97)*, pages 388–396, Los Alamitos, April 1–5 1997. IEEE Computer Society Press.

[60] Cedric Dumoulin. DREAM : A distributed shared memory model using PVM. In Hermes, editor, *EuroPVM'95*, volume 5 of *Parallelisme, reseaux et repartition*, pages 155–160, September 1995.

[61] C. Roucairol et alli. Stratageme : Une méthodologie de programmation parallèle pour les problèmes non structurés. Technical report, Rapport de Recherche, PRiSM, Versailles, December 1995.

[62] D. Dunning et alli. The virtual interface architecture. *IEEE Micro*, pages 66–76, March 1998.

[63] Joseph R. Eykholt, Steve R. Kleiman, Steve Barton, Jim Voll, Roger Faulkner, Anil Shivalingiah, Mark Smith, Dan Stein, Mary Weeks, and Dock Williams. Beyond multiprocessing : multithreading the sunOS kernel. In *Proceedings of the Summer 1992 USENIX Technical Conference and Exhibition*, pages 11–18, San Antontio, TX, USA, June 1992.

[64] G. E. Fagg, J. J. Dongarra, and A. Geist. Heterogeneous MPI application interoperation and process management under PVMPI. *Lecture Notes in Computer Science*, 1332 :91–101, 1997.

[65] G. E. Fagg, K. S. London, and J. J. Dongarra. MPIConnect : Managing heterogeneous MPI applications interoperation and process control. *Lecture Notes in Computer Science*, 1497 :93–96, 1998.

[66] B. Fagin and A. Despain. The performance of parallel Prolog programs. *IEEE Transactions on Computers*, 39(12) :1434–1445, December 1990.

[67] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3) :233–244, September 1994.

[68] D. Feldcamp and A. Wagner. Parsec : A software development environment for performance oriented parallel programming. In S. Atkins and A. Wagner, editors, *Transputer Research and Applications 6*, pages 247–262, Amsterdam, Oxford, Washington, Tokyo, May 1993. IOS Press.

[69] A. J. Ferrari and V. S. Sunderam. TPVM : distributed concurrent computing with lightweight processes. In IEEE, editor, *Proceedings of the Fourth IEEE*

International Symposium on High Performance Distributed Computing, August 2–4, 1995, Washington, DC, USA, pages 211–218, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press.

[70] S. A. Fineberg, P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO-a portable implementation of MPI-IO. In IEEE, editor, *Frontiers'96, the Sixth Symposium on the Frontiers of Massively Parallel Computation : October 27–31, 1996, Annapolis, Maryland : proceedings*, pages 188–195, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[71] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, C-21(9) :948–960, September 1972.

[72] I Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 95.

[73] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, Summer 1997.

[74] I. Foster and R. Olson. A guide to parallel and distributed programming in nperl. Technical report, Argonne National Lab, 95.

[75] I. Foster, R. Olson, and S. Tuecke. Programming in fortran M. Technical report, Argonne National Lab, 93.

[76] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1) :70–82, August 1996.

[77] Ian Foster, Robert Olson, and Steven Tuecke. Productive Parallel Programming : The PCN Approach. Technical Report MCS-P295-0392, Mathematics and Computer Science Division, Argonne National Laboratory, April 1992.

[78] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1, chapter 6 and 15. Prentice Hall, Englewood Cliffs, NJ, 1988.

[79] H. Franke, P. Hochschild, P. Pattnaik, J.-P. Prost, and M. Snir. MPI-F : an MPI prototype implementation on IBM SP1. In Jack J. Dongarra and Bernard Tourancheau, editors, *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing : Townsend, TN, USA, 25–27 May 1994*, pages 43–55, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[80] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed computing in a heterogeneous computing environment. *Lecture Notes in Computer Science*, 1497 :180–187, 1998.

[81] F. Galilee. ATHAPASCAN-*1 : Interprétation distribuée du flot de données d'un programme parallèle*. Thèse de doctorat, Institut National Polytechnique deGrenoble, Grenoble, France, September 1999.

[82] François Galilée, Jean-Louis Roch, Gerson G. H. Cavalheiro, and Mahtias Doreille. Athapascan-1 : On-line building data flow graph in a parallel language. In *Pact'98*, Paris, France, Oct 1998.

[83] T. Gautier, J-L. Roch, and G. Villard. Regular versus irregular problems and algorithms. In *Proc. of IRREGULAR'95, Lyon, France*. Springer-Verlag, Septembre 1995.

[84] Thierry Gautier. *Calcul formel et parallélisme : Conception du Système Givaro et Applications au Calcul dans les Extensions Algébriques*. PhD thesis, Institut National Polytechnique de Grenoble, France, Juin 1996.

[85] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994.

[86] Ilan Ginzburg. *Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, September 1997.

[87] William Gropp. Early experiences with the IBM SP1 and the high-performance switch. Technical Report ANL-93/41, Argonne National Laboratory, November 93.

[88] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6) :789–828, September 1996.

[89] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of chant : A talking threads package. In *Supercomputing '94*, pages 350–359, November 1994.

[90] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant : A talking thread package. Technical Report TR-94-25, Institute for Computer Applications in Science and Engineering, April 1994.

[91] Matthew Haines, Piyush Mehrotra, and David Cronk. Ropes : support for collective operations among distributed threads. Technical Report TR-95-36, Institute for Computer Applications in Science and Engineering, May 1995.

[92] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1319. Published as Proceedings of the IEEE, volume 79, number 9.

[93] R. H. Halstead. Parallel symbolic computing. *Computer*, 19(8) :35–43, August 1986.

[94] P. Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice Hall, Englewood Cliffs, 1 edition, 1977.

[95] R. W. Hockney. *The Science of Computer Benchmarking*. SIAM Publications, 1995.

[96] R. W. Hockney and I. J. Curington. $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. *Parallel Computing*, 10 :277–286, 1989.

[97] Herbert H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu. The multi-threaded architecture multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, June 1995. Also as Memo-88, McGill University, School of Computer Science, ACAPS Lab, Dec. 1994.

[98] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel : An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1) :64–76, January 1991.

[99] IEEE. *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE Computer Society Press, IEEE-Standard, New York,1993, 1993.

[100] IEEE. *IEEE 1003.1c-1994 : Standard for Information Technology — Portable Operating System Interfaces (POSIX) — Part 1 : System Application Programm Interface (API) — Amendment 2 : Threads Extension [C langage]*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994.

[101] Raj Jain. *The art of computer systems performance analysis*. John Wiley, New York, 1991.

[102] Christopher F. (Christopher Frank) Joerg. *The Cilk system for parallel multithreaded computing*. Thesis (ph.d.), Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, 1996.

[103] Geraint Jones. *Programming in occam*. Prentice-Hall international, 87.

[104] Terry Jones, Richard Mark, Jeanne Martin, John May, Elsie Pierce, and Linda Stanberry. An MPI-IO interface to HPSS. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages I :37–50, September 1996.

[105] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principle*, pages 41–55, Pacific Grove, CA, USA, October 1991. Published in ACM Operating Systems Review Vol.25, No.5, 1991. Also as Tech report CS-TR-319-91, Princeton University. Dept. of Computer Science.

[106] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming With Threads*. SunSoft Press, Mountainview, CA, USA, 1995.

[107] P. T. Koch and X. Rousset de Pina. SciOS : Flexible operating system support for SCI clusters. *Lecture Notes in Computer Science*, 1470 :601–609, 1998.

[108] Adam Kolawa and Jon Flower. Express is not just a message passing system. *Parallel Computing*, (20) :597–614, 1994.

[109] Monica S. Lam. Jade : a coarse-grain parallel programming language. In *Proceedings of the Heterogeneous Network-Based Concurrent Computing Workshop*, Tallahassee, FL, October 1991. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.91.

[110] Koen Langendoen, Raoul Bhoedjang, and Henri Bal. Models for asynchronous message handling. *IEEE Concurrency*, 5(2) :28–37, April/June 1997.

[111] Koen Langendoen, John Romein, Raoul Bhoedjang, and Henri Bal. Integrating polling, interrupts, and thread management. In *Proceedings of Frontiers '96 : The Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 13–22, Annapolis, Maryland, October 27–31, 1996. IEEE Computer Society.

[112] Bil Lewis and Daniel J. Berg. *Multithreaded programming with pthreads*. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA, 1998.

[113] Rusty Lusk and Ralph Butler. Portable parallel programming with p4. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992. Supercomputing Computations Research Institute, Florida State University.

[114] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin Theobald, and Xinmin Tian. Polling watchdog : Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 179–190, New York, May 22–24 1996. ACM Press.

[115] W. F. McColl. Bulk synchronous parallel computing. In J. R. Davy and P. M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, pages 41–63. OUP, 1995.

[116] Jean-Francois Mehaut and Raymond Namyst. Marcel : Une bibliothèque de processus légers. Technical report, laboratoire d'Informatique Fondamentale de Lille, 1995.

[117] P. Mehrotra and M. Haines. An overview of the Opus language and runtime system. *Lecture Notes in Computer Science*, 892 :346–354, 1995.

[118] MPI-2 : Extensions to the message-passing interface. The MPI Forum, July 1997.

[119] D. A. Mundie and D. A. Fisher. Parallel processing in Ada. *Computer*, 19(8) :20–25, August 1986.

[120] R. Namyst and J.-F. Méhaut. PM^2 : Parallel multithreaded machine. A computing environment for distributed architectures. In E. H. D'Hollander, G. R.

Joubert, F. J. Peters, and D. Trystram, editors, *Parallel Computing : State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, volume 11 of *Advances in Parallel Computing*, pages 279–285, Amsterdam, February 1996. Elsevier, North-Holland.

[121] Ncube Corporation. *nCUBE2 programmer's guide - revision 2*, 1990.

[122] P. Newton and J. C. Browne. The CODE 2.0 graphical parallel programming language. In ACM, editor, *Conference proceedings / 1992 International Conference on Supercomputing, July 19–23, 1992, Washington, DC*, pages 167–177, New York, NY 10036, USA, 1992. ACM Press.

[123] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-machine multicomputer : An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 224–235, May 1993.

[124] N. Nupairoj and L. M. Ni. Performance evaluation of some MPI implementations on workstation clusters. In IEEE, editor, *Proceedings of the 1994 Scalable Parallel Libraries Conference : October 12–14, 1994, Mississippi State University, Mississippi*, pages 98–105, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press.

[125] M. Oey, K. Langendoen, and H. E. Bal. Comparing kernel-space and user-space communication protocols on Amoeba. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 238–245, Los Alamitos, CA, USA, May 30–June 2 1995. IEEE Computer Society Press.

[126] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations : Illinois Fast Messages (FM) for Myrinet. In Sidney Karin, editor, *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. ACM Press and IEEE Computer Society Press.

[127] Jean-Louis Pazat. *Génération de code réparti par distribution de données.* Habilitation à diriger les recherches, Université de Rennes I, France, 1997.

[128] P. Pierce. The NX/2 operating system. In *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[129] B. Plateau, R. Jungblut, W. Stewart, and B.Ycart. Simulation performante pour les réseaux d'automates stochastiques. In *Actes de ROADEF, Deuxième congrès de la société Française de Recherche Opérationnelle et Aide à la décision*, Autrans, 1999.

[130] Brigitte Plateau and al. Présentation d'APACHE. Rapport APACHE 1, IMAG, Grenoble, October 1993.

[131] M. L. Powell, Steve R. Kleiman, Steve Barton, Devang Shah, Dan Stein, and Mary Weeks. SunOS multi-thread architecture. In *Proceedings of the Winter*

1991 USENIX Technical Conference and Exhibition, pages 65–80, Dallas, TX, USA, January 1991.

[132] L. Prylli and B. Tourancheau. BIP : A new protocol designed for high performance networking on myrinet. *Lecture Notes in Computer Science*, 1388 :472–80, 1998.

[133] R. Rabenseifner. MPI-GLUE : Interoperable high-performance MPI combining different vendor's MPI worlds. *Lecture Notes in Computer Science*, 1470 :563–571, 1998.

[134] Abhiram Ranade. A framework for analyzing locality and portability issues in parallel computing. *Lecture Notes in Computer Science*, 678 :185–193, 1993.

[135] Jeffrey Richter. Creating, managing, and destroying processes and threads under Windows NT. *Microsoft Systems Journal*, 8(7) :55–63, July 1993.

[136] Martin C. Rinard. The design, implementation and evaluation of jade : A portable, implicitly parallel programming language. Thesi CSL-TR-94-638, Stanford University, Computer Systems Laboratory, August 1994.

[137] M. Riviere. *Concepts structurants pour la mise en oeuvre d'applications irrégulières : Application au support exécutif parallèle Athapascan0mp*. Thèse de doctorat en informatique, Université Joseph Fourier, Grenoble, France, October 1997.

[138] Michiel Ronsse and Koen De Bosschere. JiTI : Tracing Memory References for Data Race Detection. In *Parallel Computing : Fundamentals, Applications and New Directions*. Elsevier, 1997.

[139] Ehud Shapiro. Concurrent Prolog : A progress report. In *IEEE Computer*, August 1986.

[140] Wei Shu and L. V. Kale. Chare kernel — a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11(3) :198–211, March 1991.

[141] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, 3 edition, 1991.

[142] Ursula Sinkewicz. A strategy for SMP ULTRIX. In USENIX Association, editor, *USENIX Conference Proceedings, Summer, 1988. San Francisco*, pages 203–212, Berkeley, CA, USA, Summer 1988. USENIX.

[143] David B. Skillicorn. *Foundations of parallel programming*. Number 6 in International series on parallel computation. Cambridge University Press, 1995.

[144] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2) :123–169, June 1998.

[145] A. Skjellum, S. G. Smith, N. E. Doss, A. P. Leung, and M. Morari. The design and evolution of Zipcode. *Parallel Computing*, 20(4) :565–596, April 1994.

[146] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6), June 1992.

[147] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI : the complete reference*. MIT Press, Cambridge, MA, USA, 1996.

[148] W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison Wesley, 1994.

[149] W. Richard Stevens. *TCP/IP Illustrated, Volume 2*. Addison Wesley, 1994.

[150] W. Richard Stevens. *UNIX Network Programming, Interprocess Communications*, volume 2. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1998.

[151] Daniel C. Swinehart, Polle T. Zellweger, and Robert B. Hagmann. The structure of Cedar. In *SIGPLAN '85 Symposium on Language Issues in Programming Environments*, pages 230–244, Seattle, Washington, June 1985. ACM.

[152] A. S. Tanenbaum. *Operating Systems : Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1987.

[153] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, New Jersey, 1992.

[154] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache : A virtual memory management technique for zero-copy communication. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 308–315, Los Alamitos, March 30–April 3 1998. IEEE Computer Society.

[155] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[156] Rajeev Thakur, Ewing Lusk, and William Gropp. Users guide for ROMIO : A high-performance, portable MPI-IO implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.

[157] Rajeev Thakur, Ewing Lusk, and William Gropp. I/O in parallel applications : The weakest link. *The International Journal of High Performance Computing Applications*, 12(4) :389–395, Winter 1998.

[158] A tightly-coupled processor-network interface. Dana S. henry and christopher F. joerg. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 27, number 9, pages 111–122, Boston, MA, October 1992.

[159] K. Ueda. Guarded horn clauses. In E. Wada, editor, *Logic Programming*, number 221 in LNCS, pages 168–179. Springer-Verlag, 1986.

[160] J. D. Ullman, Alfred V. Aho, and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974.

[161] Uresh Vahalia. *UNIX Internals*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1996.

[162] Aad J. van der Steen and Jack J. Dongarra. Overview of recent supercomputers. Technical Report UT-CS-96-325, Department of Computer Science, University of Tennessee, April 1996.

[163] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus : A flexible group communication system. *CACM*, 39(4) :76–83, April 1996.

[164] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages : A mechanism for integrated communication and computation. In David Abramson and Jean-Luc Gaudiot, editors, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–267, Gold Coast, Australia, May 1992. ACM Press.

[165] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net : a user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, volume 29, number 5, pages 303–316, 1995.

[166] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages : a mechanism for integrated communication and computation. Technical Report CSD-92-675, University of California, Berkeley, March 1992.

[167] Barry Wilkinson and Michael Allen. *Parallel Programming : Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1998.

[168] Honbo Zhou and Al Geist. LPVM : a step towards multithread PVM. *Concurrency : Practice and Experience*, 10(5) :407–416, April 1998.

Résumé

L'accroissement d'efficacité des réseaux d'interconnexion et la vulgarisation des machines multiprocesseurs permettent la réalisation de machines parallèles à mémoire distribuée de faible coût : les grappes de multiprocesseurs. Elles nécessitent l'exploitation à la fois du parallélisme à grain fin, interne à un multiprocesseur offert par la multiprogrammation légère, et du parallélisme à gros grain entre les différents multiprocesseurs. L'exploitation simultanée de ces deux types de parallélisme exige une méthode de communication entre les processus légers qui ne partagent pas le même espace d'adressage.
Le travail de cette thèse porte sur le problème de l'intégration de la multiprogrammation légère et des communications sur grappes de multiprocesseurs symétriques (SMP). Il porte plus précisément sur l'évaluation et le réglage du noyau exécutif ATHAPASCAN-0 sur ce type d'architecture. ATHAPASCAN-0 est un noyau exécutif, portable, développé au sein du projet APACHE (CNRS-INPG-INRIA-UJF), qui combine la multiprogrammation légère et la communication par échange de messages. La portabilité est assurée par une organisation en couches basée sur les standards POSIX threads et MPI largement répandus. ATHAPASCAN-0 étend le modèle de réseau statique de processus « lourds » communicants tel que MPI, PVM, etc, à celui d'un réseau dynamique de processus légers communicants. La technique de base est la multiprogrammation légère des communications et des calculs. La progression des communications exige la scrutation de l'état du réseau et l'enchaînement des opérations de transferts. L'efficacité repose sur la minimisation de ces opérations. De plus, l'emploi de multiprocesseurs ajoute des problèmes spécifiques dus à l'apparition d'un parallélisme réel entre calcul et communication. Ces problèmes sont présentés et des solutions sont proposées pour l'environnement ATHAPASCAN-0. Ces solutions sont évaluées sur des grappes de multiprocesseurs.

Mots-clés : Multiprogrammation légère, communication par échange de messages, environnement de programmation parallèle, grappes de stations, multiprocesseurs symmétriques.

Abstract

The continuous price reduction for commodity PC multiprocessors and the availability of fast network interfaces have made cluster of multiprocessors an attractive low-price alternative to build parallel systems. Multiprocessor clusters offer two levels of parallelism: a fine grain parallelism inside a single multiprocessor and a coarse grain among them. A mechanism must be provided to exploit both levels of parallelism simultaneously. This requires to provide communications between threads belonging to different addresses spaces.
This dissertation addresses the problem of integrating threads and communications on ATHAPASCAN-0 run time system. ATHAPASCAN-0 is a portable run time for cluster of multiprocessors developed as part of the APACHE project (CNRS-INPG-INRIA-UJF). Portability is achieved by a layered organization based on standards like POSIX threads and MPI. The ATHAPASCAN-0 run time system extends the heavy-weight process communication model of message passing libraries such as MPI, PVM, etc, into a lighter dynamic network of communicating threads. Multiprogramming is the key concept used. Communication progress is based on a network polling basis to handle incoming messages and to deliver outgoing communications requests. Performance is strongly dependent on the way these operations are implemented. Additionally, multiprocessors introduce some programming problems like overhead of cache coherency mechanisms, method of managing concurrent accesses and efficient mutex locking to avoid unnecessary context switching. These problems are analyzed and solutions are implemented in the ATHAPASCAN-0 run time system. An evaluation of these solutions is performed on a cluster of multiprocessors.

Keywords: Multithreading, message passing, parallel programming environnements, network of workstations, symmetric multiprocessors.