Athapascan-0b : intégration efficace et portable de multiprogrammation légère et de communications

Ilan Ginzburg

▶ To cite this version:

Ilan Ginzburg. Athapascan-0b : intégration efficace et portable de multiprogrammation légère et de communications. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1997. Français. NNT : . tel-00004946

HAL Id: tel-00004946
https://theses.hal.science/tel-00004946v1

Submitted on 20 Feb 2004

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

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

Thèse

présentée par

Ilan Ginzburg

Pour obtenir le grade de Docteur

de l'Institut National Polytechnique de Grenoble

(Arrêté ministériel du 30 mars 1992)

Spécialité : Informatique

Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications

Date de soutenance : 12 septembre 1997

Composition du jury
Jacques Briat, *responsable de thèse*
Jean-Marc Geib, *président du jury et rapporteur*
Jean-François Méhaut, *examinateur*
Brigitte Plateau, *directeur de thèse*
Thierry Priol, *rapporteur*

Thèse préparée au sein du
Laboratoire de Modélisation et de Calcul (LMC)

Si cette thèse n'est signée que de mon nom, il est pourtant clair que son contenu est une œuvre collective.

Tout a commencé par un DEA et je remercie Jacques Briat d'avoir accepté de m'encadrer alors qu'il savait que j'avais déjà programmé en COBOL. Il m'a été ensuite d'une grande aide pendant la thèse, présent et disponible (y compris dimanches et jours fériés) quand j'avais besoin de lui, tout en me laissant une grande liberté.

Brigitte Plateau m'a aidé à dégager les grandes orientations de cette thèse. Ses capacités de travail et de synthèse sont étonnantes. Merci à elle d'avoir aussi su me redonner de la motivation quand celle-ci s'essoufflait un peu…

Merci ensuite à Marcelo Pasin, qui outre le fait de m'avoir supporté pendant près de trois ans, a été d'une grande efficacité dans le développement d'Athapascan-0b et très enrichissant lors de nos nombreux échanges. C'est à lui que l'on doit entre autres la stabilisation d'Athapascan-0b et la documentation étoffée.

Un grand merci à Messieurs Jean-Marc Geib, Jean-Francois Méhaut et Thierry Priol d'avoir accepté de rapporter sur cette thèse malgré les contraintes d'emploi du temps. Leurs remarques m'ont été utiles.

Ces remerciements ne seraient pas complets sans citer les autres membres de l'équipe APACHE sans lesquels ce travail aurait été moins agréable et moins productif. Michel Christaller et Michel Rivière qui ont toujours accepté de partager leurs expériences "Athapascanesques", Jean-Louis Roch qui par ses idées a contribué à orienter (et re-orienter et re-re-orienter…) Athapascan-0b vers ce qu'il est aujourd'hui et ceux qui par leur présence ont rendu agréables (et moins longues) ces quelques années de thèse : Eric Maillet, Frédéric Guinand, Benhur Stein, François Galilée, Paulo Fernandes et tous les autres…

Pour finir, merci à la société Hewlett-Packard qui en ignorant superbement mes candidatures répétées m'a permis (malgré moi ?) de continuer et de finir cette thèse, et merci à la société Intel, qui en acceptant au contraire ma candidature m'a rendu presque agréable la phase de rédaction.[1]

Enfin, la thèse n'étant pas tout, merci à Karine, à ma famille et à mes amis pour le reste.

Ilan Ginzburg,
Grenoble, Septembre 1997.

1. Vous pouvez m'y joindre à l'adresse `gilan@iil.intel.com`.

Table des matières

1 Introduction **11**
1.1 Objectif de la thèse . . . 11
1.2 Structure du document . . . 14

2 Communication **15**
2.1 Introduction . . . 15
2.2 Définitions . . . 16
2.3 Bibliothèques de communication . . . 17
2.3.1 NX . . . 17
2.3.2 Express . . . 18
2.3.3 Parmacs . . . 18
2.3.4 P4 . . . 18
2.3.5 Zipcode . . . 19
2.4 PVM . . . 19
2.4.1 Présentation . . . 19
2.4.2 Architecture . . . 20
2.5 MPI . . . 21
2.5.1 Présentation . . . 21
2.5.2 Exemple d'architecture . . . 24
2.6 Discussion . . . 25

3 Multiprogrammation légère **27**
3.1 Introduction . . . 27
3.2 Caractéristiques d'un processus léger . . . 28
3.3 Raisons d'utiliser les processus légers . . . 29
3.4 Fonctionnalités des processus légers POSIX . . . 31
3.4.1 Création, vie et mort . . . 31
3.4.2 Synchronisation . . . 33
3.4.3 Données privées et pile . . . 33
3.4.4 Signaux . . . 34
3.4.5 Annulation . . . 34
3.5 Processus légers et le système d'exploitation . . . 35
3.5.1 N vers 1 (niveau utilisateur) . . . 36

3.5.2 1 vers 1 (niveau noyau) 37
3.5.3 N vers P (N > P) 38
3.6 Ordonnancement 39
3.6.1 Ordonnancement FIFO 39
3.6.2 Ordonnancement *round-robin* 40
3.6.3 Autres ordonnancements 40
3.6.4 Priorités et *mutex* 40
3.7 Processus légers et codes existants 42
3.7.1 Compatibilité d'une fonction avec la multiprogrammation légère 42
3.7.2 Causes possibles des problèmes 43
3.8 Bilan 44

4 Multiprogrammation et communications 45
4.1 Introduction 45
4.2 Définitions 46
4.3 Systèmes existants 47
4.3.1 TPVM 47
4.3.2 Nexus 49
4.3.3 Chant 50
4.3.4 MPI-F 52
4.3.5 PM2 52
4.3.6 DTMS 53
4.3.7 Athapascan-0a 54
4.4 Résumé 55
4.4.1 Modèle de programmation 55
4.4.2 Implantation 56
4.5 Bilan 57

5 Présentation d'Athapascan-0b 61
5.1 Objectif 61
5.2 Structure et concepts 63
5.2.1 Tâches 64
5.2.2 Services 64
5.2.3 Démarrage et initialisation 65
5.2.4 Fils et synchronisation 65
5.2.5 Ports 65
5.2.6 Tampons 66
5.2.7 Types de données 66
5.2.8 Opérations bloquantes et asynchrones 66
5.2.9 Ordonnancement 67
5.3 interface de programmation 67
5.4 Fonctionnalités 69
5.4.1 Initialisation et terminaison 69

5.4.2 Création et utilisation de tampons 70
5.4.3 Déclaration et appel de service 71
5.4.4 Création locale de processus légers 72
5.4.5 Synchronisation locale 73
5.4.6 Création de port 73
5.4.7 Envoi et réception de messages 74
5.5 Démarche générale d'utilisation 75

6 Réalisation 77
6.1 Problématique 77
6.1.1 Objectif 77
6.1.2 États d'un fil d'exécution communiquant 77
6.1.3 Intégration efficace 79
6.2 Objectifs 82
6.3 Architecture 83
6.3.1 Athapascan-0b 84
6.3.2 Akernel 86
6.4 Avancement des communications 89
6.4.1 Utilisation monoprogrammée 89
6.4.2 Utilisation multiprogrammée 90
6.5 Validation des choix 91

7 Performance d'Athapascan-0b 93
7.1 Introduction 93
7.2 Fonctionnalités de base 93
7.2.1 Création locale d'un fil d'exécution 94
7.2.2 Commutation de processus léger 98
7.2.3 Envoi et réception d'un message 100
7.2.4 Appels de service 106
7.3 Fonctionnalités composées 108
7.3.1 Recouvrements 108
7.3.2 Appel et retour parallèles 112
7.3.3 Échange total 115
7.4 Bilan 118

8 Conclusion et perspectives 119

Table des figures

2.1 Deux modes de communication entre tâches PVM. 20
2.2 Séparation des communications par les communicateurs MPI 23

3.1 Processus lourds monoprogrammés et multiprogrammés. 28
3.2 Transitions d'état d'un fil d'exécution. 32
3.3 Plusieurs fils d'exécution sont liés à une entité noyau. 36
3.4 Chaque fil d'exécution est lié à une entité noyau. 37
3.5 Un ensemble de fils d'exécution se partage un ensemble d'entités noyau. 38
3.6 Inversion de priorité non bornée sur un monoprocesseur. 41

5.1 Deux tâches dans une machine parallèle Athapascan-0b. 64

6.1 Transitions d'état d'un fil d'exécution effectuant des communications. 78
6.2 Communication d'un processus lourd. 81
6.3 Communication d'un processus léger. 81
6.4 Communication de plusieurs processus légers. 82
6.5 Architecture d'Athapascan-0b. 83

7.1 Temps de création d'un processus léger DCE et Athapascan-0b sur AIX 3.2, fonction du nombre de processus légers déjà présents. 95
7.2 Deux modes de création d'un processus léger DCE sur AIX 3.2. . . . 95
7.3 Temps de création d'un processus léger *Pthreads* ou Athapascan-0b sur AIX 4.2, fonction du nombre de processus légers déjà présents. . . 96
7.4 Perturbation du temps de création d'un processus léger *Pthreads* sur AIX 4.2. 97
7.5 Temps de commutation de fils DCE ou Athapascan-0b sur AIX 3.2, fonction du nombre de processus légers présents. Ordonnancement à partage de temps sans priorités. 98
7.6 Temps de commutation de fil DCE ou Athapascan, fonction du nombre de threads présents. Ordonnancement FIFO ou *round-robin* avec priorités. 99
7.7 Temps d'envoi de messages, Athapascan-0b et MPI-F sur AIX 3.2. . . 101
7.8 Temps d'envoi de petits messages, Athapascan-0b et MPI-F sur AIX 3.2.101

7.9 Diagramme temporel d'un “ping-pong” pour messages de taille nulle en Athapascan-0b sur AIX 3.2. 104
7.10 Temps d'appel de service, de service urgent et d'envoi. Athapascan-0b sur AIX 3.2. 107
7.11 Temps total d'un échange “ping-pong” avec plusieurs fils communicants par nœud. Athapascan-0b sur AIX 3.2. 109
7.12 Temps total d'un échange “ping-pong” avec plusieurs fils communicants par nœud – gros messages. Athapascan-0b sur AIX 3.2. 109
7.13 “Ping-pong” classique et “ping-pong” dans lequel les messages sont envoyés simultanément des deux nœuds. MPI-F sur AIX 3.2. 110
7.14 Durée de recouvrement sur le nœud “ping” pendant un échange “ping-pong” complet (envoi et réception). Athapascan-0b sur AIX 3.2. . . . 111
7.15 Appel de n services par un fil et récupération des résultats. 112
7.16 Appel de n services et récupération des résultats, petits paramètres et résultats. Athapascan-0b sur AIX 3.2. 113
7.17 Appel de n services et récupération des résultats. Athapascan-0b sur AIX 3.2. 114
7.18 Appels de services, échange total entre les services puis renvoi de résultats. 115
7.19 Appels de services, échange total puis renvoi de résultats. Athapascan-0b sur AIX 3.2. 116
7.20 Propagation du retard de création de service lors d'un échange total entre deux services. 117

8.1 Visualisation de traces prises lors de l'exécution de l'application de dynamique moléculaire. 120

Chapitre 1

Introduction

L'utilité d'architectures parallèles et distribuées est aujourd'hui reconnue par la communauté informatique. En effet, la puissance des processeurs ainsi que les débits des réseaux de communication augmentent rapidement mais les besoins des applications plus vite encore.

Les technologies existantes telles les bibliothèques de communication (PVM [89, 47, 24, 46, 68], MPI [38, 25, 37]) et les directives de partitionnement de données (HPF [67]) permettent l'exploitation efficace du parallélisme matériel (l'ordinateur parallèle ou distribué) quand les calculs peuvent être divisés en sous-calculs de poids égal, avec une bonne localité des données. Les applications de ce type sont dites **régulières**.

Dans le cas contraire, quand le calcul ne peut être divisé en sous calculs de taille équivalente ou que la localité des données des sous calculs ne peut être assurée, l'application est dite **irrégulière**. Afin de supporter efficacement l'exécution de telles applications, le support d'exécution (matériel et logiciel) doit permettre l'exécution à faible surcoût de tâches à grain fin (c'est-à-dire dont les calculs ont une durée du même ordre de grandeur que la durée d'une "petite" communication) et proposer un mécanisme permettant d'équilibrer la charge des différents processeurs.

Une approche possible consiste à proposer un **réseau dynamique de processus légers communicants** comme support d'exécution d'applications irrégulières. Elle permet de répondre simplement et efficacement aux problèmes soulevés par les applications irrégulières dans la mesure où le faible surcoût des processus légers permet une exécution efficace de tâches à grain fin et l'application peut être découpée en un nombre de tâches supérieur au nombre de processeurs disponibles, ceci permettant un équilibrage de charge dynamique simple.

1.1 Objectif de la thèse

La multiprogrammation légère (*multi-threading*) est de plus en plus utilisée dans tous les domaines de l'informatique (voir pourquoi en section 3.3, page 29). Cette utilisation sans cesse croissante, dans les applications comme dans les systèmes d'exploi-

tation, a conduit à la standardisation d'une interface utilisateur de multiprogrammation légère ainsi qu'à une disponibilité de celle ci sur la plupart des plates-formes actuelles (POSIX *threads*, voir section 3.1, page 27).

Similairement, avec la popularisation des applications parallèles tournant sur machines parallèles dédiées ou sur réseaux de stations de travail, se sont développées des bibliothèques de communication entre processus. Leur standardisation n'est pas aussi avancée que celle de la multiprogrammation légère. Au long de l'"histoire" des bibliothèques de communication, certaines se sont imposées comme standards de fait. Un standard d'interface de bibliothèque a également été proposé par un comité et est assez largement accepté aujourd'hui (MPI, voir section 2.5, page 21 et suivantes). On trouve des bibliothèques se conformant à ce standard sur un grand nombre de plates-formes.

Contrairement à la grande disponibilité de ces deux composants, les bibliothèques associant ces deux aspects, à savoir mariant multiprogrammation légère et communications, ne sont pas encore très répandues et il n'existe pas de standardisation à ce sujet. Pourtant, l'utilisation conjointe de communications et de multiprogrammation légère semble intéressante, par exemple par les possibilités qu'elle offre de masquer les temps de communication par des calculs utiles et de gérer simplement et élégamment l'indéterminisme des communications. Certaines plates-formes proposent une telle association de communications et de multiprogrammation légère, avec des interfaces de programmation "propriétaires" : les applications qui y sont développées ne sont pas portables vers d'autres systèmes. D'autres bibliothèques ont été portées et sont disponibles sur plusieurs systèmes (voir chapitre 4, page 45 et suivantes).

L'intérêt de l'utilisation combinée des communications et de la multiprogrammation légère semble aujourd'hui acquis et de nombreux groupes s'intéressent ou se sont intéressés à ce problème. Un des objectifs fixés au départ de cette thèse alors que les environnements associant communications et multiprogrammation légère étaient moins courants qu'actuellement, était de montrer l'utilité d'une telle combinaison de fonctionnalités : au delà de la résolution des problèmes techniques posés par l'"assemblage" des fonctionnalités, montrer l'utilité d'une utilisation conjointe de communications et de multiprogrammation légère.

L'utilisation d'un tel support de programmation, associant communication et multiprogrammation légère, est nécessaire pour le développement d'applications utilisant ces fonctionnalités. En effet, la complexité de sa mise en œuvre est relativement importante et ses fonctionnalités sont du niveau de celles d'un système d'exploitation. Il ne serait pas économiquement ou fonctionnellement intéressant que chaque programmeur recrée un tel environnement dans son application.

Différentes stratégies ou choix techniques sont possibles pour le développement d'un tel support d'exécution. Chacune présente bien sur des avantages et des inconvénients. Le support de programmation peut être développé en partant de rien (*from scratch*) et optimisé pour une plate-forme ou architecture particulière. Le produit résultant est performant et adapté à la machine mais les temps de développement et de mise au point sont longs et la portabilité compromise : un portage se traduit souvent par un nouveau développement. C'est cette stratégie qui est souvent adoptée par les construc-

teurs pour proposer un environnement de communication et de multiprogrammation légère sur leurs machines.

Un autre choix consiste à utiliser un ou des produits “standard”, disponibles sur la plupart des architectures, en les modifiant pour adapter leurs fonctionnalités. Le temps de développement est plus court que dans le cas précédent mais nécessite de disposer du code source du produit modifié pour **chacune** des architectures cibles. Ceci n’est pas toujours aisé surtout quand le produit en question est un développement propriétaire du constructeur de la machine. Un autre inconvénient est qu’il faut modifier chaque nouvelle version de la bibliothèque ou du programme “standard” utilisé afin de lui intégrer les modifications.

Le troisième choix, celui défendu dans cette thèse, consiste à construire cet environnement de communication et de multiprogrammation légère en n’utilisant que des logiciels “standard” disponibles sur un grand nombre de machines, **en ne modifiant pas leur code**. La thèse défendue est la suivante :

> L’intégration de logiciels standard par assemblage extérieur, c’est-à-dire sans modification de ces logiciels, est une solution viable au problème d’un support d’exécution mariant communications et multiprogrammation légère. Cette façon de faire présente un bon compromis entre la **performance**, la **portabilité** et les **fonctionnalités**. Un tel support d’exécution permet une programmation plus simple et plus efficace pour certains types de problèmes.

Une telle approche d’intégration présente des avantages certains :

- Temps de développement et de mise au point relativement courts.
- Très bonne portabilité (quasi instantanée) sur les plates-formes disposant des outils standard utilisés. La plupart des plates-formes disposent de ces outils car ce sont des standards.
- Utilisation sans modification de logiciels ayant fait leurs preuves et débarrassés d’un grand nombre de bogues.
- Il n’est pas nécessaire de disposer du code source des logiciels utilisés. Ceci permet d’utiliser les logiciels propriétaires développés par les constructeurs, souvent plus efficaces que les versions “domaine public” correspondantes.
- Prise en compte rapide et facile des progrès des logiciels standard. En effet, le logiciel étant utilisé sans modifications, une nouvelle version peut simplement se substituer à une ancienne dans l’environnement de programmation.
- Les fonctionnalités des logiciels standard sont “remontables” dans le support de programmation sans grands efforts.

Parmi les inconvénients de cette stratégie il en est un d'ordre fonctionnel, à savoir qu'il n'est pas possible d'offrir dans le support d'exécution des fonctionnalités impossibles à construire au dessus des logiciels utilisés. Par contre, ces fonctionnalités pourront être facilement "remontées" à l'utilisateur si elles sont ultérieurement intégrées dans les logiciels utilisés.

L'autre inconvénient majeur concerne la performance, dans la mesure où le "mariage" de deux logiciels indépendants (communication et multiprogrammation légère) peut ne pas donner d'aussi bonnes performances qu'un développement intégrant au plus bas niveau ces fonctionnalités.

On essayera dans cette thèse de montrer que la baisse de performance reste "raisonnable", que les fonctionnalités offertes couvrent une large classe d'applications et que les bénéfices de cette approche sont grands.

1.2 Structure du document

Le reste de ce document est structuré comme suit : les chapitres 2 et 3 présentent l'état de l'art dans les domaines des bibliothèques de communication et de la multiprogrammation légère. Le chapitre 4 présente quelques intégrations de communications et de multiprogrammation légère. Le chapitre 5 présente Athapascan-0b : ses fonctionnalités et son interface de programmation. Le chapitre 6 pose le problème d'une intégration efficace des communications et de la multiprogrammation légère et décrit la réalisation d'Athapascan-0b. Le chapitre 7 consiste en une évaluation d'Athapascan-0b. On y trouvera une évaluation de la performance "brute" d'Athapascan-0b ainsi qu'une justification (a posteriori) du bien fondé de l'utilisation conjointe de la multiprogrammation légère et des communications à travers les bénéfices qu'elle apporte à des schémas algorithmiques classiques. Une conclusion et des perspectives sont enfin présentées au chapitre 8.

Tout au long de ce document, les mots à mettre en valeur (définitions, points essentiels...) sont imprimés **en gras**, les termes étrangers *en italique* et les extraits de code et noms de fonctions façon `machine à écrire`.

Chapitre 2

Communication

2.1 Introduction

Les besoins en communications sont une des principales différences entre les ordinateurs parallèles et séquentiels. Les développeurs d'applications parallèles ont besoin de boîtes à outils de communication efficaces et portables [18]. Efficaces pour les performances et portables afin de proposer leurs applications sur des architectures différentes et d'en garantir la pérennité quand les plates-formes évoluent.

Les systèmes d'exploitation classiques proposent de longue date des primitives de communication telles que les *sockets* et leurs extensions type RPC (appel de procédure à distance). Ces primitives ont atteint un bon niveau de standardisation et sont présentes sur un grand nombre de systèmes. Elles répondent aux besoins des applications ayant besoin de services distribués simples tels les échanges client-serveur ou maître-esclave.

L'arrivée du parallélisme a quelque peu bousculé ce statu quo. Les besoins en communication sont devenus plus variés et plus exigeants en performance. Les "anciennes" interfaces ne suffisent plus et affichent des performances trop faibles sur les nouveaux réseaux de communication. En effet, le nombre de couches logicielles traversées et les recopies effectuées induisent une pénalité faible sur les anciens réseaux mais beaucoup plus sensible sur les réseaux modernes et rapides.

Des bibliothèques de communication pour le calcul parallèle ont alors commencé à se développer [70]. D'abord spécifiques à une architecture donnée, souvent développées par le constructeur de la machine, puis avec un plus grand effort de portabilité et de standardisation. Ces bibliothèques, outre des fonctionnalités mieux adaptées au calcul parallèle (voir section 2.2), présentent pour certaines la possibilité d'être implantées de façon efficace pour exploiter la latence et le débit des réseaux utilisés.

L'évolution actuelle des applications (dans le domaine du parallélisme et d'autres), vers plus de coopération et plus de communications, laisse à penser que l'évolution des bibliothèques de communication n'est pas finie.

2.2 Définitions

Sont présentées ici les définitions de termes utilisés dans les descriptions des bibliothèques de communication. Ces définitions couvrent plus ou moins la plupart des fonctionnalités trouvées dans ces bibliothèques de communication pour le calcul parallèle.

Tâche : Environnement d'exécution sur la machine parallèle comprenant un espace mémoire, l'accès à des ressources système et du temps processeur. Les tâches d'une machine parallèle communiquent entre elles et sont souvent implantées sous la forme de processus lourds.

nœud : processeur de la machine parallèle. Selon les bibliothèques et les machines, un nœud peut accueillir une ou plusieurs tâches.

Communication point à point : le fait pour deux tâches particulières d'échanger des messages entre elles. Cette forme de communication suit le modèle des processus communiquants [58].

Communication collective : un certain nombre de tâches effectue une opération de communication. Un exemple de communication collective est la diffusion d'un message par une tâche vers d'autres tâches.

Opération collective : opération impliquant un certain nombre de tâches. Chaque tâche fournit une ou plusieurs valeurs qui contribuent au calcul d'un résultat final. Par exemple l'addition de toutes les valeurs contribuées par les tâches (opération de réduction). Une autre forme d'opération collective n'implique pas d'échange de données et est utilisée pour la synchronisation, comme par exemple les barrières (attente que toutes les tâches aient atteint un certain point de leur exécution).

Communication synchrone et asynchrone : dans une communication synchrone ou bloquante, lorsque l'appel à la primitive de communication retourne, la communication est terminée (le sens de "terminée" dépend de la primitive). Dans une communication asynchrone ou non bloquante, le retour de l'appel ne signifie pas sa terminaison qui intervient plus tard.

Communication par interruption : forme particulière de communication asynchrone dans laquelle le système d'exploitation (ou la bibliothèque de communication) signale à l'application la fin de la communication par une interruption.

Étiquette : un message étiqueté ou typé (*tagged message*) contient une valeur particulière (étiquette ou *tag*) dans son en-tête qui permet au récepteur d'effectuer des réceptions sélectives. Des messages avec des étiquettes différentes échangés entre deux tâches constituent autant de liaisons indépendantes.

Réception sélective : dans le cas des message étiquetés, réception limitée aux messages dont l'étiquette a une valeur donnée.

Tampon : dans certains systèmes, préalablement à l'envoi d'un message, les données doivent être recopiées dans un tampon (emballage) qui est alors envoyé. La réception se fait alors de façon symétrique, le tampon est reçu puis les données en sont recopiées vers leur emplacement définitif (déballage).

Topologie : dans les machines parallèles dans lesquelles les nœuds ne sont pas tous reliés entre eux (ou accessibles directement les uns depuis les autres), on parle de topologie pour décrire les connections existantes.

Hétérogène : se dit d'une machine parallèle dans laquelle les nœuds ne sont pas tous du même type. Il peut s'agir de processeurs d'architectures différentes ou encore de systèmes d'exploitation différents.

Latence : temps qui s'écoule pour installer une communication, sans compter le temps de transfert des données. Dans le modèle classique où la durée d'une communication est une fonction affine de la taille des données envoyées, la latence est la durée de l'envoi d'un message de taille nulle.

Débit : quantité de données pouvant être transférées par unité de temps par une communication. Le débit est souvent mesuré en octets par seconde et peut varier suivant les types de communication et la taille des messages envoyés.

2.3 Bibliothèques de communication

Quelques exemples de bibliothèques et environnements de communication sont donnés dans cette section. La bibliothèque PVM et le standard MPI sont traités de façon plus détaillée dans des sections séparées (et comparés dans [48]).

2.3.1 NX

NX [78, 79, 77] est l'interface de programmation des ordinateurs parallèles Intel (notamment iPSC/860 et Paragon). Les objectifs de NX sont de cacher la topologie de la machine au programmeur et permettre à deux tâches quelconques de communiquer simulant ainsi un réseau complètement maillé. NX permet également de placer plusieurs tâches sur chaque nœud. NX offre des opérations globales, des communications asynchrones (avec attente explicite de fin de communication ou signalisation par une interruption exécutant une fonction fournie par l'application) et l'étiquetage des messages pour des réceptions sélectives.

2.3.2 Express

Express [75, 36] est une boîte à outils commercialisée par la société ParaSoft. Son modèle de programmation parallèle comprend les primitives classiques de communications point à point, des opérations globales et des opérations de distribution de données. Express propose également des primitives adaptées à la gestion de graphiques par une application parallèle en multiplexant les ordres de dessin des différents nœuds sur une seule fenêtre d'affichage. Les divers outils composant Express gèrent différents aspects du développement de programmes parallèles. La philosophie d'utilisation d'Express consiste à partir d'un programme séquentiel pour le paralléliser par étapes jusqu'à arriver à un programme parallèle efficace. On trouve dans Express un outil permettant de visualiser l'exécution de programmes séquentiels, un autre permettant d'analyser l'accès aux données de programmes séquentiels ou parallèles, un troisième effectuant une parallélisation automatique de code ainsi qu'un débogueur parallèle.

2.3.3 Parmacs

Parmacs [13] (*Parallel Macros*) est un ensemble de macros constituant un système d'échange de messages. Le projet est né à l'Argonne National Laboratory, a été repris au GMD (Allemagne) et est aujourd'hui un produit commercialisé par la société allemande Pallas. L'exécution d'une application Parmacs commence par une tâche (processus hôte) dont le rôle principal est la création des autres tâches de la machine parallèle. Les tâches sont créées simultanément et le seul moyen de modifier la composition de la machine parallèle consiste à toutes les détruire et en créer de nouvelles. Les tâches disposent de primitives de communication synchrones et asynchrones. Les messages échangés sont des zones contiguës de mémoire et sont étiquetés par une valeur entière. Les réceptions peuvent se faire en sélectionnant le type et l'identité de la tâche émettrice des messages à recevoir. Parmacs permet de décrire des topologies (anneaux, tores, grilles et graphes quelconques) et d'optimiser le placement des tâches sur les nœuds. Les fonctionnalités topologiques de MPI (voir [38]) ont été fortement inspirées de Parmacs.

2.3.4 P4

La bibliothèque p4 [12], issue des débuts du projet Parmacs, a pour objectif de permettre le développement portable d'une grande variété d'algorithmes. À ce titre, elle supporte aussi bien les machines à mémoire partagée à travers des moniteurs [56, 57], que les machines à mémoire distribuée à travers l'échange de messages. P4 offre sur les machines à mémoire partagée un ensemble de moniteurs ainsi que les primitives permettant d'en construire de nouveaux. P4 possède également des mécanismes de gestion des tâches à travers des fichiers de configuration de la machine parallèle qui permettent une organisation hiérarchique en grappes (*clusters*) selon le modèle maître-esclave. Les primitives d'échange de messages comprennent les classiques envois et

réceptions de messages en versions synchrones et asynchrones, avec gestion de l'hétérogénéité ainsi que des opérations globales.

2.3.5 Zipcode

Zipcode [84] est un système d'échange de messages visant à fournir un support au développement de bibliothèques parallèles. L'objectif est de séparer les messages internes échangés par les bibliothèques des messages du code utilisateur. Zipcode introduit trois concepts à cet effet : les groupes de tâches servant à définir les domaines des communications et opérations collectives, les contextes servant à isoler différents plans de communication et les *mailers*, associant groupes, contextes et topologies pour former des espaces de communication. Zipcode a influencé MPI sur ces thèmes et les communicateurs de MPI (présentés en section 2.5) sont semblables aux *mailers*.

2.4 PVM

2.4.1 Présentation

PVM, acronyme de *Parallel Virtual Machine*, est un environnement de programmation développé à l'Oak Ridge National Laboratory [89, 47, 24, 46, 68]. PVM introduit le concept de **machine virtuelle** qui a révolutionné le calcul distribué hétérogène en permettant de relier des machines différentes pour créer une machine intégrée parallèle. Son objectif est de faciliter le développement d'applications parallèles constituées de composants relativement indépendants s'exécutant sur une collection hétérogène de machines, comme par exemple un ensemble de stations de travail. Les machines participant au calcul peuvent être des monoprocesseurs, des multi-processeurs ou des machines spécialisées, permettant ainsi aux composantes du calcul de s'exécuter sur l'architecture la plus adaptée à l'algorithme.

PVM offre les communications point à point étiquetées entre tâches et des communications et opérations globales. Les communications se font exclusivement par l'intermédiaire de tampons, dans lesquels sont emballées les données. Les envois sont synchrones mais les réceptions peuvent être synchrones ou asynchrones : il est possible de tester une réception sans se bloquer ou encore se bloquer en réception pour une durée de temps spécifiée.

Pour le support des communications et opérations collectives, PVM introduit la notion de groupe. Les groupes sont dynamiques, il est possible à une tâche de s'insérer dans un groupe et d'en sortir. Entre membres d'un groupe, il est possible de faire des diffusions, des barrières ainsi que des opérations collectives de réduction par des fonctions fournies par l'utilisateur.

L'ensemble des tâches s'exécutant sur l'ensemble des nœuds constitue la machine virtuelle PVM. Elle est dynamique car PVM permet l'adjonction de nœuds pendant l'exécution ainsi que la création de nouvelles tâches sur les nœuds. Il en est de même

pour la suppression de tâches ou de nœuds en cours d'exécution. Enfin, PVM supporte l'hétérogénéité des machines.

On ne peut considérer PVM comme une simple bibliothèque de communication. C'est plutôt un support complet pour un **réseau dynamique de processus lourds communicants** (les tâches PVM sont des processus lourds).

2.4.2 Architecture

PVM est composé de deux parties principales :

- Le démon PVM,
- La bibliothèque de routines, constituant l'interface de programmation de PVM.

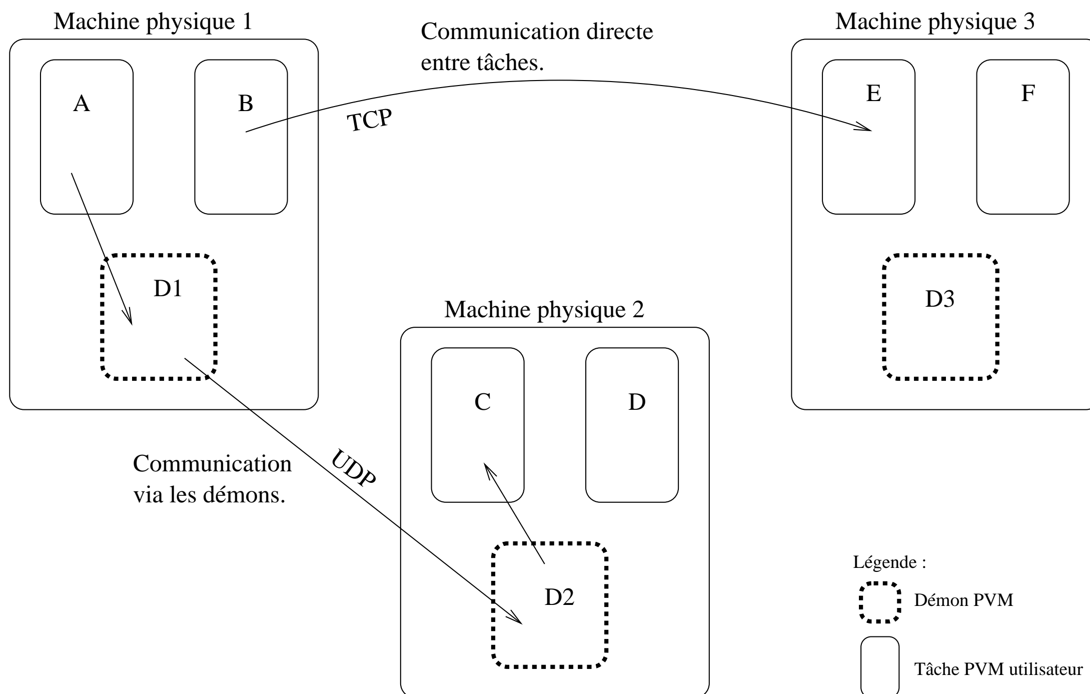

FIG. 2.1 – *Deux modes de communication entre tâches PVM.*

Le démon PVM est un processus lourd présent sur chaque nœud participant à la machine virtuelle et sert de point de contact sur la machine, en étant globalement connu par les tâches de la machine virtuelle (concept de *well-known service*). La première communication entre deux tâches de machines physiques différentes se fait via les démons : une tâche désirant communiquer entre en contact avec son démon local, qui communique avec le démon du nœud distant qui achemine le message jusqu'à la tâche destinataire. Les communications suivantes peuvent se faire directement entre les tâches, sans faire intervenir les démons, quand les tâches se sont mis d'accord sur

les ports de communication. Les démons de la machine virtuelle communiquent entre eux par le protocole UDP [80] (fiabilisé par le démon), les tâches utilisateur qui communiquent directement utilisent TCP [81]. Le nombre de connexions TCP simultanées est souvent limité sur une machine, il n'est donc pas possible la plupart du temps d'ouvrir une connexion TCP par tâche avec laquelle une tâche communique, ce qui fait que toutes les tâches ne peuvent employer ce moyen de communication. La communication entre une tâche et son démon local se fait par le protocole le mieux adapté disponible sur la machine. La figure 2.1 illustre les deux modes de routage, par les deux indirections via les démons et directement entre deux tâches. Le routage direct a bien sur l'avantage de la rapidité car le nombre de recopies du message est réduit.

La deuxième composante de PVM est la bibliothèque de fonctions utilisable par le programmeur des tâches. Cette bibliothèque constitue l'API (*Application Programming Interface*) de PVM. Elle contient un ensemble fonctionnellement complet de primitives nécessaires à la coopération entre tâches d'une application, à savoir l'envoi et la réception de messages, la gestion de groupes de tâches, la création de processus à distance, la coordination entre tâches et la modification de la machine virtuelle. Les fonctionnalités relatives aux groupes et aux communications et opérations collectives sont également intégrées dans la bibliothèque et construites au dessus des primitives de base de PVM. La bibliothèque s'occupe notamment de la gestion des tampons de communication, de l'emballage et du déballage des données dans ces tampons et de l'étiquetage des messages.

2.5 MPI

2.5.1 Présentation

MPI [38, 25, 37, 92], acronyme de *Message Passing Interface*, est un standard définissant la syntaxe et la sémantique d'un noyau de routines destinées à l'écriture de programmes parallèles portables. Dans MPI ont été intégrées les fonctionnalités les plus intéressantes de plusieurs systèmes existants. Cette standardisation a entre autres objectifs de permettre des communications hautement efficaces et ce sur différentes plates-formes, de proposer une interface compatible avec une large palette de systèmes qui ne soit pas trop éloignée des pratiques courantes et finalement de ne pas interdire des implantations comportant de multiples fils d'exécution.

Le standard définit les communications point à point, les communications et opérations collectives, les groupes de tâches, les communicateurs, les topologies de processus, la syntaxe des appels en C et en Fortran 77, l'interface de prise de trace et instrumentation et les fonctions relatives à l'environnement d'exécution (horloge etc. . .). Par contre, d'importants aspects de la programmation parallèle ne sont pas abordés, à savoir les opérations relatives à la mémoire partagée, la délivrance de messages par interruptions [1] (*interrupt-driven messages*), l'exécution à distance et les messages actifs

1. Il s'agit de signaler à l'application l'arrivée d'un message pour lequel elle a déjà posté (effectué)

(messages déclenchant un traitement à leur arrivée sur le site distant), la multiprogrammation légère, la gestion des machines et des processus (tâches) et les opérations d'entrée sortie.

MPI propose les concepts de **groupe**, **contexte** et **communicateur**. Les groupes permettent de définir les participants à des communications et opérations collectives et de structurer l'exécution de l'application en permettant à des groupes différents d'exécuter des codes différents. Ils permettent aussi d'adapter les schémas de communication à la topologie de la machine parallèle. Les contextes permettent de créer des plans distincts de communication et de s'assurer que les communications sur différents plans ne se mélangent pas. Cette séparation est utile quand différentes parties d'une même application communiquent entre elles, comme par exemple quand un programme parallèle communiquant utilise une bibliothèque parallèle qui elle même communique entre ses différents éléments. Les communicateurs sont enfin le moyen d'utiliser les contextes lors de l'envoi et la réception de messages.

Les communications point à point s'appuient sur les concepts précédents. Un message est envoyé à une tâche d'un rang donné dans un groupe, à l'aide d'un communicateur donné (l'indication du groupe est contenue dans le communicateur). Le message est marqué d'une étiquette. Lors d'une réception, ces paramètres permettent d'effectuer un filtrage sur les messages à recevoir : si le communicateur doit être explicitement spécifié, il est par contre possible d'utiliser des jokers (*wildcard*) pour l'indication du rang de la tâche émettrice dans le groupe et pour la valeur de l'étiquette.

La figure 2.2 montre un exemple de programme utilisant deux bibliothèques. Chacune des bibliothèques ainsi que le code utilisateur échangent des messages entre les tâches. Ces messages sont reçus par le bon destinataire (code de la bibliothèque concernée ou code utilisateur) car des communicateurs différents sont utilisés. Sans communicateurs, les bibliothèques et le code utilisateur auraient dû s'entendre sur un étiquetage non ambigu des messages, ce qui pose de sérieux problèmes quand on utilise des bibliothèques existantes ou que les codes ne sont pas développés conjointement.

Il existe trois modes de communication :

- Mode **standard** : Il n'est pas nécessaire que la tâche destinataire ait posté (effectué) la réception au moment où la tâche émettrice poste l'envoi. Le message sera délivré plus tard, quand la réception aura été postée. L'envoi peut terminer sur la tâche émettrice **avant** que la réception ne soit postée (les données sont alors tamponnées).

- Mode **ready** : La réception doit être postée avant l'envoi. Dans le cas contraire, l'issue est indéterminée.

- Mode **synchronous** : L'opération d'envoi attend que la réception correspondante ait été postée et ne termine pas avant.

---

une réception.

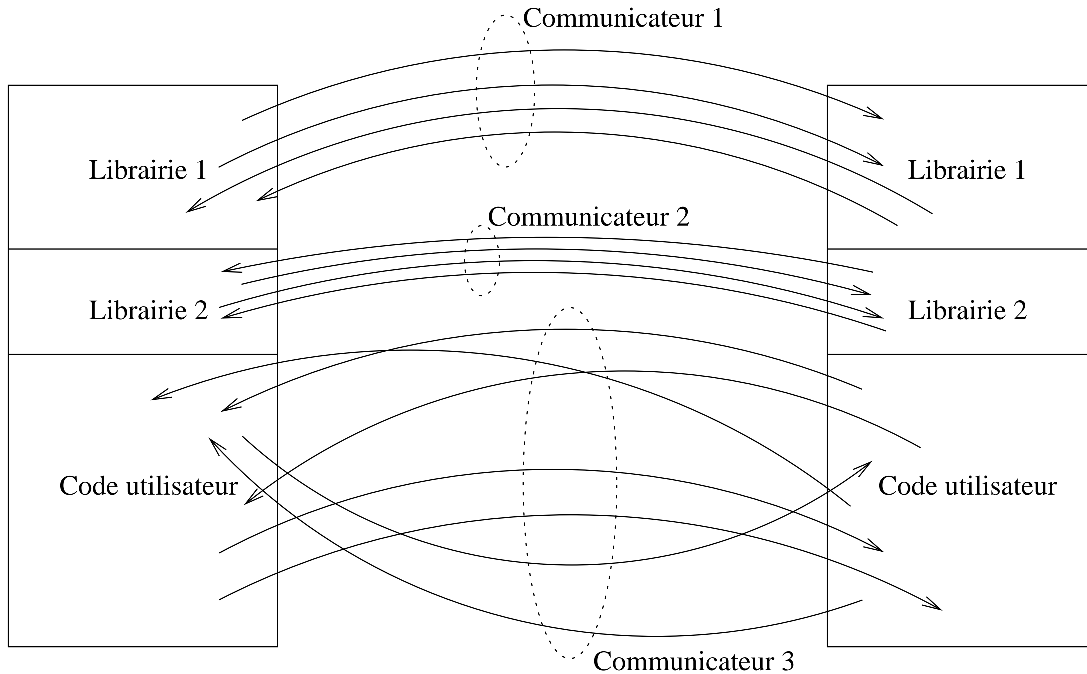

FIG. 2.2 – *Séparation des communications par les communicateurs MPI*

Chacun de ces trois modes de communication peut se décliner de deux façons différentes : primitive synchrone ou asynchrone. Une primitive synchrone ne retourne que quand les paramètres de l'appel (tampons etc...) peuvent être réutilisés par l'application sans compromettre l'envoi. Une primitive asynchrone retourne immédiatement et renvoie un pointeur (*handle*) sur une structure de communication permettant de tester ultérieurement la terminaison de l'opération ou de se bloquer sur son attente. Il existe donc six primitives d'envoi différentes. Le choix des primitives de réception est plus restreint, il n'en existe que deux versions, à savoir la réception synchrone et la réception asynchrone, qui fonctionne de manière similaire à la primitive d'envoi asynchrone. Même si la primitive de réception asynchrone retourne immédiatement, la réception n'est effectivement terminée que quand les données auront été mémorisées à l'endroit spécifié.

Les données envoyées dans les messages sont typées, et MPI propose un mécanisme pour définir le type et la structuration en mémoire des données (*datatypes*). Ce mécanisme permet d'envoyer de façon (potentiellement) efficace des données non contiguës en mémoire par l'utilisation de mécanismes matériels adaptés et permet à MPI de disposer des informations nécessaires sur les données afin d'effectuer leur conversion pour gérer l'hétérogénéité des machine et des réseaux.

En plus des communications point à point, MPI offre des communications et opérations collectives. Les communications collectives concernent tous les membres d'un groupe et peuvent être des diffusions personnalisées ou pas (*[personalized] one to all*), des échanges totaux personnalisés ou pas (*[personalized] all to all*). Les opérations

collectives sont les réductions et les barrières. Une communication ou une opération collective doit être appelée avec des paramètres compatibles par tous les membres du groupe. Les réductions et les communications collectives n'impliquent pas nécessairement de barrière (au choix de l'implantation). Les opérations et communications collectives ne sont disponibles qu'en version bloquante.

2.5.2 Exemple d'architecture

Plusieurs groupes ont proposé des implantations se conformant au standard MPI, la plus connue d'entre elles étant MPICH[2] [50] développée à l'Argonne National Laboratory. D'autres implantations ont été proposées dont LAM/MPI [10] par le Ohio Supercomputer Center (cette version a été utilisée pour la plupart des portages d'Athapascan-0b, voir section 6.5 page 91), CHIMP-MPI [9] par le Edinburgh Parallel Computing Center ainsi que des implantations "propriétaires" développées par IBM [43, 45], SGI, Convex, Intel, Meiko, Cray et DEC.

MPICH est implantée au dessus d'une couche nommée ADI : *Abstract Device Interface* ou "interface du dispositif abstrait[3]" [49]. On distingue donc deux couches : la couche basse (ADI) et la couche haute, appelée API (*Application Programming Interface*) et qui dans le cas présent est MPICH, mais pourrait être autre. L'ADI est à son tour implantée sur des mécanismes de plus bas niveau, bibliothèques standard : TCP, P4, Nexus, *Channel Interface*[4]... ou propriétaires : NX (Intel), MPL (IBM)...

Cette structure en couches de MPICH, pouvant utiliser des bibliothèques présentes sur la plupart des architectures, lui permet d'être portée très rapidement sur une nouvelle architecture, quitte à optimiser le portage par la suite. Elle permet aussi de choisir le niveau auquel se fait le portage et qui conditionne la quantité de travail nécessaire et les performances (les choix possibles sont le portage de l'ADI sur bibliothèque standard, sur bibliothèque propriétaire optimisée pour la plate-forme ou encore réécriture de l'ADI optimisée pour la machine).

ADI

Les fonctionnalités de l'ADI se répartissent en quatre groupes : spécification des messages à envoyer et à recevoir, déplacement de données entre le programme utilisateur et le réseau physique, gestion des files d'attente (queues) des messages reçus et mise à disposition d'informations sur l'environnement d'exécution. Quand le matériel utilisé (le "dispositif") ne fournit pas certaines fonctions (comme par exemple gestion des files d'attente), celles-ci sont gérées par logiciel. Cette approche permet de tirer pleinement profit des fonctionnalités offertes par une architecture donnée tout en

2. Le "CH" dans le nom représente le caméléon (*chameleon*) dont l'adaptation à l'environnement et la rapidité de mouvement de la langue symbolisent les objectifs de portabilité et d'efficacité de MPICH et qui est également le nom d'une interface de bas niveau utilisée par MPICH.

3. On fait ce qu'on peut...

4. Dont Chameleon, un ensemble de macros, en est l'implantation la plus célèbre.

garantissant une portabilité vers des architectures plus simples. Pour atteindre cet objectif, les échanges entre l'ADI et l'API se font via des macros. Chacune des couches définit des macros qui sont utilisées par l'autre. De cette façon il est possible de tester à la compilation si une fonctionnalité de l'ADI est disponible et d'adapter l'exécution en conséquence (simulation d'une fonctionnalité absente ou utilisation d'une fonctionnalité présente).

Le transfert des données entre l'API et l'ADI peut se faire de plusieurs manières, dans l'objectif d'éviter autant que faire se peut des recopies coûteuses. L'API et l'ADI peuvent manipuler des structures de données éparpillées en mémoire afin d'obtenir les meilleures performances d'un dispositif (*device*) qui en serait également capable. L'ADI spécifie, toujours à l'aide de macros, quelles sont les types de structures de données qu'elle peut gérer directement. Une paquétisation des messages est également disponible, afin de ne pas imposer aux deux couches l'utilisation de tampons non bornés.

Deux files d'attente se trouvent dans l'ADI (ou dans le dispositif). Celle des réceptions de messages non encore satisfaites, ainsi que celle des messages arrivés mais pour lesquels il n'y a pas encore eu de réception. L'ADI comme l'API manipulent potentiellement ces deux files, il y a donc un problème d'accès concurrents. La solution adoptée consiste à effectuer toutes les manipulations des files depuis l'ADI, l'API lui envoyant ses requêtes quand nécessaire.

L'ADI a également pour rôle d'interfacer l'API à l'environnement extérieur. À ce titre, elle fournit des renseignements tels que le nombre de tâches dans la machine parallèle, l'horloge etc… L'ADI est également responsable du découpage des messages en paquets et de l'adjonction des en-têtes, de la gestion des différentes politiques de tamponnement des messages et du support d'architectures hétérogènes.

2.6 Discussion

Il est important pour une bibliothèque de communication de pouvoir servir de support au développement d'applications parallèles efficaces et utilisant les techniques modernes de programmation (objets, multiprogrammation légère…). Pour remplir ce rôle, la bibliothèque doit avoir certaines propriétés permettant de construire efficacement les fonctionnalités nécessaires :

Support d'envois et de réceptions efficaces. Ceci implique par exemple que la bibliothèque ne doit pas imposer des copies de données qui pourraient être évitées. En particulier, ne pas nécessiter l'emballage systématique des données dans des tampons.

Présence de primitives d'envoi et de réception asynchrones. L'asynchronisme permet d'augmenter l'utilisation des ressources en ne bloquant pas inutilement une exécution. De plus, les primitives synchrones se construisent facilement au dessus des asynchrones.

Étiquetage des messages et réceptions sélectives. L'étiquetage permet d'avoir plusieurs flots de communication indépendants entre deux tâches. La réception est facilitée et rendue plus efficace dans la mesure où un message est directement reçu par la partie du programme à laquelle il est adressé, en évitant des recopies et des décodages coûteux. L'avantage est encore plus net quand le récepteur comprend plusieurs fils d'exécution.

Il peut être intéressant d'offrir les communications collectives comme primitive de base de la bibliothèque de communication, pour permettre de tirer profit d'un réseau physique disposant de la diffusion. De tels réseaux ne sont pas courants dans les machines parallèles, et habituellement les communications et opérations collectives sont construites au dessus des communications point à point.

Parmi les systèmes décrits, PVM est le plus complet dans le sens où en plus des primitives de communication il permet la création dynamique de tâches. Cette création de tâche est lourde, car elle nécessite la création d'un processus lourd, opération coûteuse. Elle n'est donc pas adaptée aux calculs à grain moyen ou fin. De plus, les primitives d'échange de message de PVM sont également lourdes, car elles nécessitent des recopies pouvant être évitées (emballage et déballage des messages).

Le standard MPI est quant à lui conçu avec un souci d'efficacité. Il propose des primitives pouvant être implantées de façon "légère" pour ne pas pénaliser des exécutions sur réseaux rapides. Les aspects de création de tâche ne sont par contre pas traités.

Une bibliothèque de communication peut être "complétée" par de la multiprogrammation légère pour proposer un moyen efficace de communication et de création dynamique de tâches. On obtiens alors les avantages de PVM sans en subir les inconvénients. Le reste de ce document présente les processus légers puis leur "mariage" avec une bibliothèque de communication.

Chapitre 3

Multiprogrammation légère

3.1 Introduction

La multiprogrammation légère (*multithreading*) consiste à exécuter plusieurs processus légers[1] au sein d'un processus lourd. Un processus lourd est l'entité d'exécution de base offerte par le système d'exploitation qui garantit une isolation et une protection par rapport aux autres processus lourds du système. Dans un processus lourd "normal" ou monoprogrammé, on ne trouve qu'un seul fil d'exécution. L'exécution du code ne se fait donc qu'en un seul endroit (qui évolue au cours du temps, bien sur). Par contre, dans un processus lourd multiprogrammé, l'exécution se fait à plusieurs endroits en même temps et ces différentes exécutions, ou différents **fils d'exécution**, partagent les mêmes ressources, celles du processus lourd. Une représentation imagée consisterait à voir un processus lourd comme un terrain de golf. Dans le premier cas, sans multiprogrammation, un seul joueur "exécute" le processus en jouant sur le terrain. Dans le deuxième cas, plusieurs joueurs jouent en même temps. Des problèmes peuvent surgir si les joueurs se gênent en utilisant les mêmes ressources au même moment (deux joueurs jouant le même trou simultanément) et dans un terrain de golf comme dans un processus multiprogrammé, une certaine forme de synchronisation et d'entente préalable entre les acteurs est nécessaire. Dans la figure 3.1, un seul fil d'exécution ou *thread* s'exécute dans le processus lourd $P1$. Dans le processus lourd $P2$ plusieurs fils s'exécutent simultanément et ont accès à la même mémoire et aux mêmes ressources, celles du processus lourd $P2$. De nombreux livres traitant des processus légers ont vu le jour ces dernières années [62, 65, 74, 64, 11].

Il existe une norme pour l'interface de programmation des processus légers. Il s'agit du standard POSIX 1003.1c-1995 défini par l'IEEE PASC (*Institute of Electrical and Electronics Engineers Portable Application Standards Committee*) et approuvé en juin 1995 [60]. Ce standard est généralement appelé POSIX *threads* ou *Pthreads*. La plupart des grands constructeurs d'ordinateurs en proposent une implantation sur leurs machines. Un brouillon antérieur de ce standard a également été implanté et a connu

1. "processus léger", "fil d'exécution", "fil", "flot d'exécution", "flot" et "*thread*" sont synonymes.

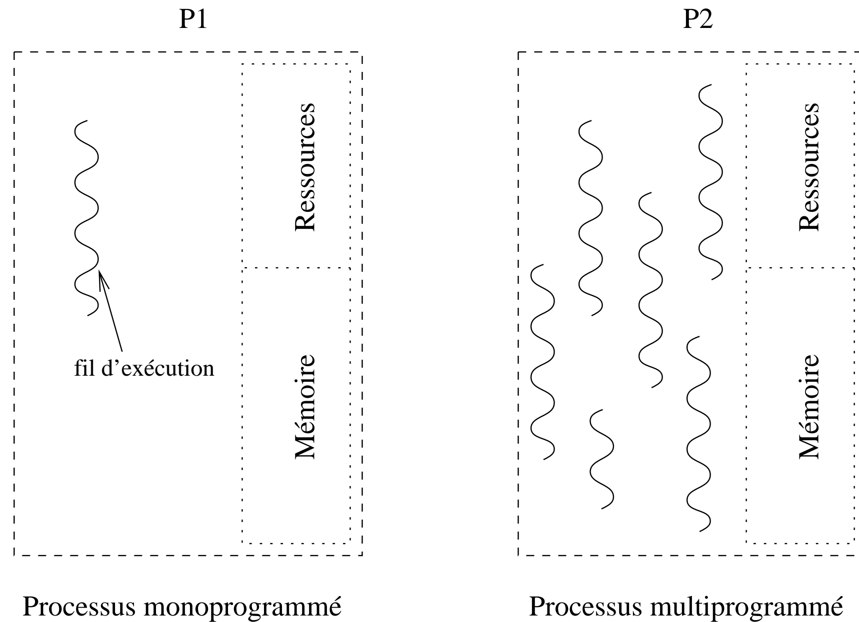

FIG. 3.1 – *Processus lourds monoprogrammés et multiprogrammés.*

une large diffusion. Il s'agit du *draft* 4 de POSIX 1003.4a de 1990 (1003.4a/D4) [59], présenté dans [22], qui a été intégré au *Distributed Computing Environment* (DCE) [5] de l'*Open Software Foundation* (OSF) et est plus généralement connu sous le nom de DCE *threads*.

Ce chapitre présente les processus légers et les raisons menant à leur utilisation, les fonctionnalités principales des processus légers POSIX, l'ordonnancement des processus légers et leur interaction avec le système d'exploitation et enfin les problèmes pouvant être rencontrés lorsque des codes "classiques" sont confrontés à la multiprogrammation légère.

3.2 Caractéristiques d'un processus léger

Le qualificatif "léger" a été donné au processus léger car comparativement à un processus lourd, il ne consomme que très peu de ressources [23, 15, 72, 35, 1]. Chaque processus lourd implique la gestion d'un nouvel espace d'adressage virtuel et de nouvelles copies de toutes les variables et ressources nécessaires à l'exécution (pile, registres, fichiers ouverts, verrous etc...). Chaque nouveau processus léger ne nécessite qu'une nouvelle pile et un nouveau jeu de registres. Les autres ressources sont partagées entre tous les processus légers s'exécutant dans un même processus lourd (le code exécuté n'est dupliqué ni quand plusieurs processus lourds exécutent le même code ni bien sur quand plusieurs processus légers exécutent le code d'un même processus lourd. L'utilisation de processus légers n'entraine donc pas d'économies sur ce point).

Cette "légèreté" ou économie en ressources a deux conséquences principales : une commutation de processus légers est 10 à 100 fois plus rapide qu'une commutation de processus lourds et il n'y a plus de protection ou isolation des ressources des processus légers [1].

Des temps de commutation très courts autorisent l'utilisation de processus légers distincts pour des tâches relativement peu coûteuses (courte durée d'exécution), auxquelles il serait impossible de dédier des processus lourds, le coût de gestion de ceux-ci (coûts de création et de terminaison) rendrait l'exécution très inefficace. L'absence de protection entre processus légers et d'isolation de leur ressources peut être un inconvénient. Si cela est effectivement le cas dans certains contextes, il en est d'autres ou ce partage des ressources est un avantage : il facilite la coopération entre ces processus. Ces deux aspects combinés engendrent une façon de programmer dans laquelle les différentes fonctionnalités de l'application sont traitées par des processus (légers, bien sur…) distincts et plus simples [31].

3.3 Raisons d'utiliser les processus légers

Les caractéristiques des processus légers rendent leur utilisation intéressante sur plusieurs aspects :

Débit Lorsqu'un processus monoprogrammé effectue un appel bloquant au système d'exploitation, le processus est bloqué tant que le service n'est pas terminé. Dans un processus multiprogrammé, seul le fil d'exécution ayant effectué l'appel se bloque, les autres pouvant continuer leur exécution.

Prenons l'exemple d'un programme qui fait des appels de procédure à distance. Dans une version monoprogrammée, le programme doit faire un appel puis attendre le résultat avant de continuer avec l'appel suivant. Dans une version multiprogrammée, chaque appel est pris en charge par un fil d'exécution différent. Même si sur un monoprocesseur les appels seront faits en séquence, il n'y aura ensuite qu'une seule période d'attente, d'où un gain de temps d'exécution.

Multi-processeurs Un processus multiprogrammé pourra utiliser efficacement une machine multi-processeur SMP (*Symmetric Multi Processor*, soit une machine dans laquelle tous les processeurs ont un même accès à la mémoire qui est donc partagée) [15]. Son exécution sera accélérée car plusieurs fils d'exécution pourront s'exécuter en parallèle, chacun sur un autre processeur. Un tel processus s'exécutera correctement sur une machine monoprocesseur également, où les fils s'exécutent à tour de rôle sur un seul processeur.

Temps de réponse La qualité d'une application interactive dépend en grande partie de sa vitesse de réaction aux sollicitations de l'utilisateur. Si dans une telle application les opérations longues (demandées par l'utilisateur ou initiées par l'applica-

tion) sont traitées par des fils d'exécution séparés, les fils d'exécution assurant l'interaction avec l'utilisateur peuvent assurer une bonne réactivité [54].

Il en est de même pour les serveurs qui peuvent recevoir de multiples requêtes simultanément. Si chacune de ces requêtes est prise en charge par un fil d'exécution séparé, les requêtes longues ne bloquent pas les requêtes courtes. Il est également possible de gérer des priorités sur le traitement des requêtes à l'aide de priorités sur les fils d'exécution.

Évitement d'interblocages L'utilisation de multiples fils d'exécution peut éviter des interblocages dans certaines configurations. Par exemple, un serveur de base de données fait appel à un serveur de noms qui l'utilise à son tour pour le stockage de ses données. Si le serveur de données ne peut traiter la requête du serveur de noms avant d'avoir fini la requête qu'il lui a adressée, il en résultera un interblocage des deux serveurs (ou du moins un blocage du serveur de données). Par contre, si le serveur de données est multiprogrammé, il prendra en compte la requête du serveur de noms alors que la première requête n'a pas terminée, ce qui permettra à cette requête d'aboutir. [2]

Facilité de programmation Les points énumérés ci-dessus peuvent être traités sans la multiprogrammation, en écrivant un programme à un seul fil d'exécution qui simule un ensemble d'exécutions indépendantes. C'est ce qui est fait dans les processus monoprogrammés traitant des événements : une boucle principale prend connaissance des nouveaux événements à traiter puis appelle en séquence, l'une après l'autre, les fonctions correspondant au traitement de chaque événement. L'écriture d'un tel programme est fastidieuse pour peu qu'on ait besoin de garder un contexte d'exécution entre deux traitements consécutifs d'un même type d'événement ou que le traitement de certains événements soit trop long et qu'il faille le découper en plusieurs petits traitements (pour ne pas bloquer les autres événements trop longtemps).

Le multiprogrammation facilite l'écriture de tels programmes. Chaque type d'événement est traité par un fil d'exécution spécifique n'ayant qu'une tâche simple à réaliser et dont la programmation se trouve simplifiée [31, 51].

La multiprogrammation a de nombreux avantages, comme nous venons de le voir, mais il y a un revers à cette médaille [66]. Les difficultés introduites par la multiprogrammation concernent notamment la synchronisation des différents fils d'exécution et la protection des accès aux données. Il faut s'assurer que l'ordre d'exécution des différentes parties du programme est compatible avec l'algorithme et que les données ne vont pas être corrompues par des accès concourants par différents fils d'exécution. Il faut également choisir habilement le niveau de découpe en fils d'exécution.

2. Les processus légers peuvent également introduire des interblocages quand une suite de verrouillages n'est pas faite correctement comme par exemple dans le cas où deux fils possèdent un verrou chacun et se bloquent tous les deux en attente du verrou possédé par l'autre fil.

Une découpe trop grossière ne permet pas de bénéficier des avantages de la multiprogrammation et une découpe trop fine est trop pénalisante pour les performances (la multiprogrammation légère a un coût : création et commutation de fil, protection des accès aux données, eventuels effets de cache etc...). Enfin, la multiprogrammation peut n'être d'aucun bénéfice pour certaines applications. Par exemple un programme de calcul intensif s'exécutant sur un monoprocesseur ne tirerait pas profit de l'utilisation de fils d'exécution multiples mais en payrait le surcoût.

L'utilisation de la multiprogrammation légère ne semble enfin pas devoir être limitée au logiciel. Les futurs microprocesseurs la proposeront sûrement sous une forme ou une autre [28, 85, 90].

3.4 Fonctionnalités des processus légers POSIX

La norme des processus légers POSIX est largement acceptée aujourd'hui et la plupart des constructeurs en proposent une implantation sur leurs machines. Les fonctionnalités de ces processus légers sont d'autre part assez semblables, ne serait-ce que dans leurs principes, à celles d'autres noyaux de multiprogrammation légère. C'est pourquoi il a été choisi de présenter plus particulièrement les fonctionnalités des POSIX *threads*.[3]

3.4.1 Création, vie et mort

Un processus léger est créé par un appel au noyau de multiprogrammation.[4] Puisqu'il faut bien commencer quelque part, lors du lancement d'un processus, un premier fil d'exécution existe déjà. Il correspond au fil d'exécution unique d'un processus monoprogrammé (voir figure 3.1). Ce premier fil d'exécution peut en créer d'autres, qui à leur tour pourront en créer d'autres etc. Il dispose d'un statut un peu particulier, dans la mesure où dans la plupart des cas le processus (lourd) abritant tous ces fils d'exécution prendra fin au moment où le fil initial se termine. Mis à part ceci, et le fait que la taille de la pile du fil initial peut être modifiée au cours de son exécution, ce qui n'est pas le cas pour les autres fils, tous les fils d'un processus lourd ont des caractéristiques semblables.

Un fil exécute une fonction donnée, admettant un paramètre et renvoyant un résultat. Lors de la création d'un nouveau fil, le fil créateur spécifie la fonction à exécuter ainsi que le paramètre qui lui est passé. La primitive de création renvoie au fil créateur l'identificateur du fil nouvellement créé. Contrairement à un appel de fonction classique, qui termine avant que l'exécution ne passe à l'instruction suivante chez l'appelant, la primitive de création d'un nouveau fil retourne avant que le fil termine (c'est bien là l'intérêt des fils d'exécution...). Un fil désirant recueillir le résultat renvoyé par

3. le fait qu'Athapascan-0b est implanté au dessus de ces processus légers n'est pas non plus totalement étranger à cette décision...

4. Synonyme ici de "bibliothèque de processus légers".

un autre fil doit en attendre la terminaison grâce à une primitive spécifique à laquelle il passe l'identificateur du fil à attendre (primitive `join`). Les ressources utilisées par un fil ayant terminé son exécution et dont le résultat a été attendu peuvent être libérées. Dans le cas où le résultat renvoyé par un fil ne sera pas attendu et que l'on souhaite que les ressources utilisées par ce fil soient libérées dès qu'il termine, il faut faire une opération de détachement (primitive `detach`) sur ce fil. Il est alors impossible d'attendre la terminaison du fil (voir [62] pour une description de ces primitives).

Après sa création, et tant que les ressources utilisées par un fil n'ont pas été libérées, un fil se trouve dans un des quatre états dont la description suit, et dont les transitions sont illustrées par la figure 3.2.

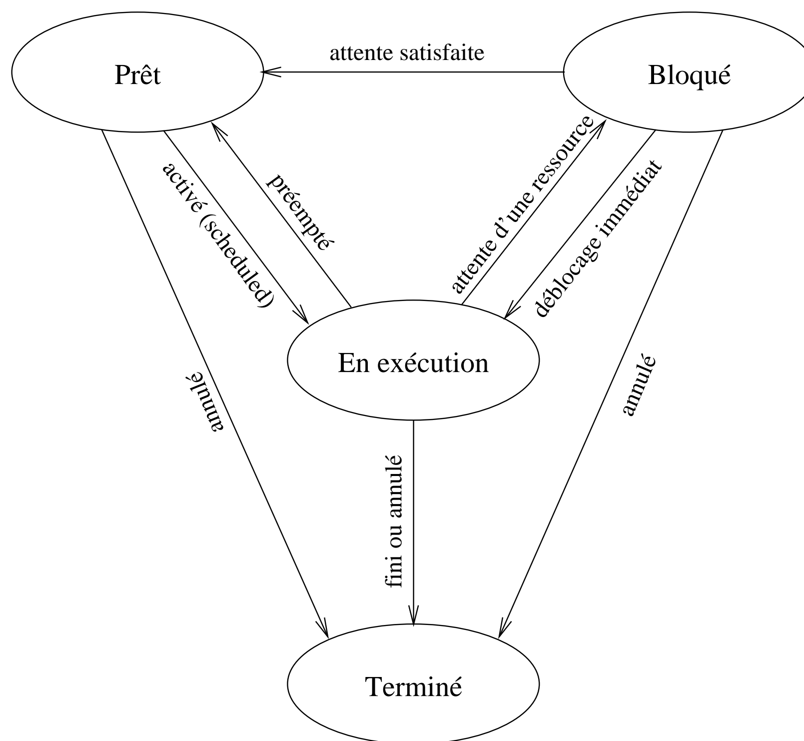

FIG. 3.2 – *Transitions d'état d'un fil d'exécution.*

Prêt Le fil est prêt à être exécuté mais n'est pas en cours d'exécution. C'est le cas d'un fil venant d'être créé, d'un fil venant d'être débloqué ou lorsqu'un ou plusieurs autres fils occupent le ou les processeurs disponibles.

En exécution Le fil est en cours d'exécution sur un processeur. Sur une machine multi-processeur, plusieurs fils peuvent être dans cet état au même moment.

Bloqué Le fil ne peut pas s'exécuter car il est en attente. Attente sur une synchronisation ou la fin d'une opération d'entrée sortie par exemple.

Terminé Le fil a terminé l'exécution de sa fonction ou a été annulé (*cancelled*), n'a pas été détaché et n'a pas encore été attendu. Dès qu'il sera attendu ou détaché, ses ressources seront libérées et il disparaîtra.

3.4.2 Synchronisation

Différents fils d'exécution doivent synchroniser leurs accès à des données ou des ressources partagées afin d'en maintenir la cohérence [61]. Les noyaux de processus légers offrent généralement deux primitives pour la synchronisation, *mutex* et *condition variable* [22, 62].

Mutex Un *mutex* (verrou d'exclusion mutuelle ou simplement verrou) a deux états : verrouillé ou non verrouillé. Quand il n'est pas verrouillé, un fil d'exécution peut le verrouiller afin de protéger l'exécution d'une section critique dans son code. En fin de section critique, le fil ayant verrouillé le *mutex* le déverrouille. Quand un fil d'exécution essaye de verrouiller un *mutex* qui l'est déjà, soit il est bloqué en attendant que le *mutex* soit déverrouillé, soit l'appel retourne en lui indiquant que le *mutex* n'était pas disponible.

condition variable Les variables de conditions permettent à un fil de suspendre son exécution tant que des données partagées n'ont pas atteint un certain état. Une variable de condition est utilisée conjointement avec un *mutex*. Celui-ci protège l'accès aux données partagées et la variable de condition permet d'attendre que ces données vérifient un certain prédicat. Un fil bloqué en attente sur une variable de condition est réveillé lorsque les données sont modifiées par un autre fil qui signale la variable de condition. Le fil en attente peut alors tester le prédicat et se remettre en attente si celui-ci n'est pas vérifié. Il existe également une attente limitée dans laquelle si la variable de condition n'est pas signalée par un autre fil pendant une durée déterminée, le réveil se fait automatiquement.

Au dessus de ces deux primitives peuvent être construits les outils classiques de synchronisation comme les sémaphores et les moniteurs.

3.4.3 Données privées et pile

Les différents fils d'exécution partagent l'espace mémoire du processus lourd qui les héberge. Ils ont accès à toutes les adresses mémoire de cet espace, qu'il n'est pas possible de protéger sélectivement par rapport à tel ou tel fil. Néanmoins, afin d'être en mesure de gérer des informations non partagées par tous les fils, il est possible de créer des zones de données privées (*thread-specific data*) qui sont en pratique utilisées par un seul fil [62, 82]. Ces zones sont accédées à l'aide de clés communes qui donnent accès à des données privées et non partagées entre les fils. Les clés servent à distinguer les différentes zones entre elles : chaque couple clé et identité du fil effectuant l'accès détermine une zone mémoire distincte.

Une autre zone mémoire qui n'est pas partagée entre les différents fils d'un processus lourd est la pile d'exécution de chaque fil. En effet, les exécutions des différents fils sont dans une large mesure indépendantes, la pile ne peut donc être partagée. Cette pile, comme toute pile d'exécution sert à stocker le contexte de retour lors d'un appel de fonction (adresse de retour, registres etc. . .) ainsi que les variables de ces fonctions. Si les piles des différents fils d'exécution sont allouées consécutivement en mémoire, il ne leur est pas possible de croître au delà de la taille spécifiée lors de la création du fil, et seule la pile du fil initial peut croître librement, comme celle d'un processus monoprogrammé où la pile croît vers la zone de mémoire utilisée pour le tas. Dans le cas où les piles des processus légers sont organisées en mémoire virtuelle avec suffisamment d'espace entre elles, une certaine croissance leur est permise. Il n'y a pas d'avantage à laisser de l'espace entre les zone allouées pour les piles pour permettre leur croissance. En effet, dans le cas d'une allocation d'une large zone de mémoire virtuelle, les pages de mémoire ne sont effectivement allouées (consommation de mémoire physique) qu'à partir du moment où le fil y fait un accès et génère une faute de page. Une allocation de larges zones consécutives en mémoire ou de petites zones espacées revient donc au même. Des mécanismes de protection de pages permettent d'assurer que la pile d'un fil ne déborde pas de la place qui lui est réservée (voir page 229 et suivantes de [62]).

3.4.4 Signaux

L'utilisation de signaux conjointement à la multiprogrammation légère demande une attention particulière. Chaque processus léger dans un processus lourd peut avoir son propre masquage des signaux, pour indiquer quels signaux il veut traiter et lesquels il ignore [82, 86]. Par contre les traitants asynchrones de signaux (*signal handlers*) sont définis pour l'ensemble du processus lourd. Il n'est de plus pas possible d'envoyer un signal à un processus léger particulier d'un autre processus lourd : le signal est délivré à un processus léger arbitraire du processus lourd destinataire (exception faite des signaux synchrones générés par l'exécution elle même qui sont délivrés au processus léger les ayant causés). Une façon plus adaptée d'utiliser les signaux avec la multiprogrammation légère consiste à masquer les signaux dans tous les processus légers d'un processus lourd et d'attendre explicitement un ensemble de signaux (attente bloquante).

3.4.5 Annulation

Dans certains cas, il peut être utile d'arrêter un fil d'exécution avant que celui-ci n'ait terminé son exécution. On parle alors d'annulation (*cancellation*) d'un fil par un autre fil [11]. Quand un fil est annulé, avant de terminer son exécution (état "terminé" de la figure 3.2) il exécute d'éventuels traitants de nettoyage (*cleanup handlers*). Ces traitants servent à laisser les ressources partagées utilisées par le fil (verrous etc. . .) dans un état cohérent. Un fil d'exécution peut choisir parmi trois modes de gestion des

annulations :

Annulation interdite L'annulation n'a pas lieu tant que le fil reste dans ce mode. Elle reste en suspens tant que l'annulation n'est pas autorisée (les deux modes suivants).

Annulation différée L'annulation pourra avoir lieu au prochain point d'annulation (*cancellation point*) atteint par l'exécution du fil. Un certain nombre d'appels système sont définis comme étant (ou pouvant être pour certains) des points d'annulation. L'annulation du fil ne pourra donc avoir lieu que lors de ces appels.

Annulation asynchrone L'annulation du fil peut avoir lieu à tout moment lors de l'exécution.

3.5 Processus légers et le système d'exploitation

Les processus légers séparent la notion de point d'exécution des autres aspects des processus lourds, comme la mémoire virtuelle et les entrées sorties. Le système d'exploitation gère traditionnellement tous les aspects de l'exécution d'un processus lourd, à savoir l'allocation des ressources mémoire virtuelle, l'ordonnancement de l'exécution du processus relativement aux autres processus et l'arrêt et le redémarrage d'une exécution (sauvegarde et restauration du contexte d'exécution). La gestion de l'ordonnancement des processus légers est répartie entre l'application et le système d'exmploitation. Cette répartition conditionne l'indépendance du système d'exploitation de l'implantation des processus légers et le contrôle de l'allocation des ressources processeur aux processus légers. Des questions se posent alors : comment sera fait l'ordonnancement des fils d'un processus lourd ? Ces fils se partageront-ils le temps processeur de leur processus lourd hôte, ou seront-ils ordonnancés indépendemment de leur processus hôte, relativement aux autres fils en exécution ? Les réponses à ces questions sont des choix d'implantation d'une bibliothèque de processus légers et de son intégration avec le système d'exploitation.

Un concept intermédiaire d'"entité noyau" (*kernel entity*) a été introduit pour désigner l'entité qui se voit attribuer un processeur pour exécuter tel ou tel processus léger [1]. Cette entité peut être vue comme un processeur virtuel et l'ensemble de ces entités et la mémoire virtuelle qu'elles partagent peuvent être vus comme un SMP virtuel.

Il existe plusieurs façons d'implanter des processus légers. On classe ces implantations selon l'interaction entre les processus légers et les entités noyau. On s'intéresse à la façon dont est fait le multiplexage entre processus légers et entités noyau et plus particulièrement au nombre de processus légers et au nombre d'entités noyau sur lesquelles ils sont multiplexés.

3.5.1 N vers 1 (niveau utilisateur)

Dans ce type d'implantation (figure 3.3), tous les processus légers d'un processus lourd se partagent la même entité noyau pour leur exécution [69, 32, 93]. Il s'agit d'une implantation compatible avec les "vieux" systèmes d'exploitation non prévus pour les processus légers et qui de ce fait ne disposent que d'une seule entité noyau par processus lourd monoprogrammé. Ce type d'implantation s'appelle parfois "processus légers en espace utilisateur" (*user-space threads*) car il peut être réalisé sous forme de bibliothèque, sans modifier le noyau.

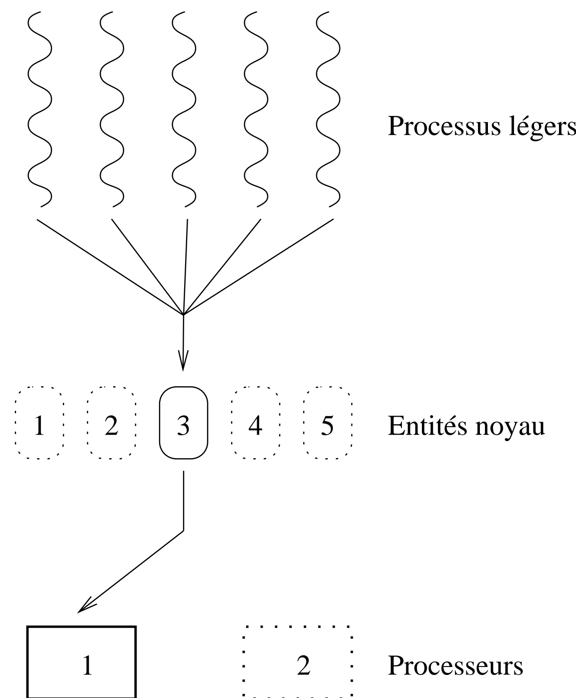

FIG. 3.3 – *Plusieurs fils d'exécution sont liés à une entité noyau.*

Une telle implantation présente des avantages dans la mesure où une commutation de fil est rapide : la sauvegarde et la restauration des registres se fait à l'intérieur du programme utilisateur, sans appels coûteux au système. De plus, un telle implantation peut être en grande partie portable sur d'autres systèmes. Le revers de la médaille est que quand un fil d'exécution fait un appel système bloquant, tous les fils du même processus lourd se bloquent. Un palliatif consiste à reprendre les opérations les plus critiques pour les rendre non bloquantes. De plus, il n'est pas possible d'exploiter plusieurs processeurs physiques pour les processus légers d'un même processus lourd, puisque l'entité noyau associée est placée sur un processeur physique donné, et les processus légers, invisibles du noyau, sont condamnés à s'y exécuter.

3.5.2 1 vers 1 (niveau noyau)

Dans cette implantation (figure 3.4), chaque processus léger est pris en charge par une entité noyau différente [29]. L'implantation des processus légers est de fait reportée totalement dans le noyau du système d'exploitation. Ce n'est donc pas sans raison que ce type d'implantation est appelé "processus légers noyau" (*kernel threads*).

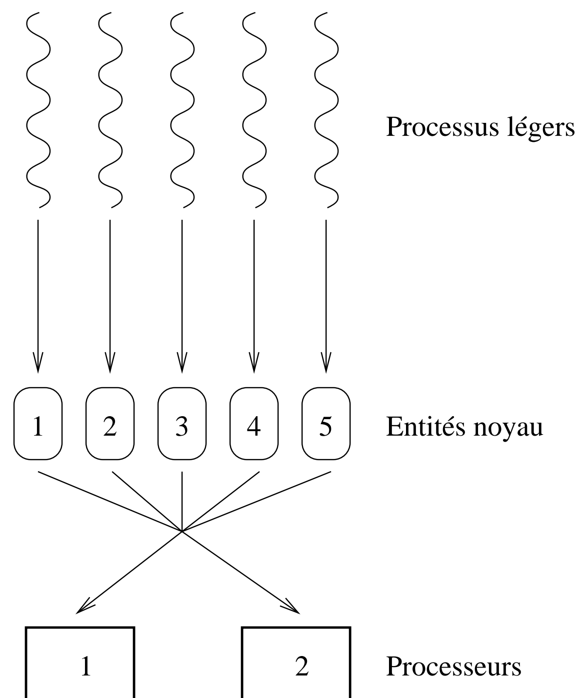

FIG. 3.4 – *Chaque fil d'exécution est lié à une entité noyau.*

Dans une telle implantation les blocages des processus légers se font dans le noyau, à travers un blocage de l'entité noyau. Le système peut alors passer à l'exécution d'une autre entité noyau, associée à un autre processus léger. On n'a plus la situation où un processus léger bloqué empêche les autres processus légers de s'exécuter : les synchronisations entre processus légers ainsi que les attentes d'entrées sorties ne bloquent que le processus léger concerné. L'implantation peut également exploiter pleinement les machines multi-processeur, puisque le système d'exploitation peut placer des entités noyau différentes sur différents processeurs physiques.

Ce type d'implantation pose des problèmes. Le premier concerne la possibilité d'étendre le système (*scalability*). Les entités noyau occupent de la place mémoire. La mémoire disponible dans le noyau n'est pas illimitée, d'autant plus que souvent les ressources noyau (telles que processus, entités système etc...) sont stockées dans des tableaux de taille fixe. Cet aspect limite donc le nombre de processus légers disponibles pour l'ensemble du système. Les ressources disponibles pour un processus donné pourront également dépendre de l'utilisation des ressources par les autres processus du système. Le deuxième problème concerne le coût élevé d'opérations telles

que la commutation de fil ou la synchronisation. Elles doivent se faire à l'aide d'appels système, coûteux car ils impliquent un changement de contexte et des vérifications par le noyau de la validité des paramètres. De plus, de tels processus légers noyau doivent intégrer l'ensemble des fonctionnalités raisonnablement nécessaires à l'ensemble des applications, qui peuvent être pénalisantes en performance pour les applications n'en ayant pas besoin (ordonnancement préemptif par exemple).

3.5.3 N vers P (N $>$ P)

Ce troisième type d'implantation (figure 3.5) tente de réconcilier les deux types précédents, en n'en gardant que les avantages [1, 82, 29, 14, 86]. L'idée est de disposer d'une implantation de processus légers en niveau utilisateur ayant à sa disposition plusieurs entités noyau. On obtient alors le meilleur des deux mondes : l'implantation en niveau utilisateur garantit des temps de commutation et de synchronisation très courts et la multiplicité des entités noyau permet d'éviter les blocages généralisés quand un fil se bloque et autorise l'exploitation efficace des multi-processeurs.

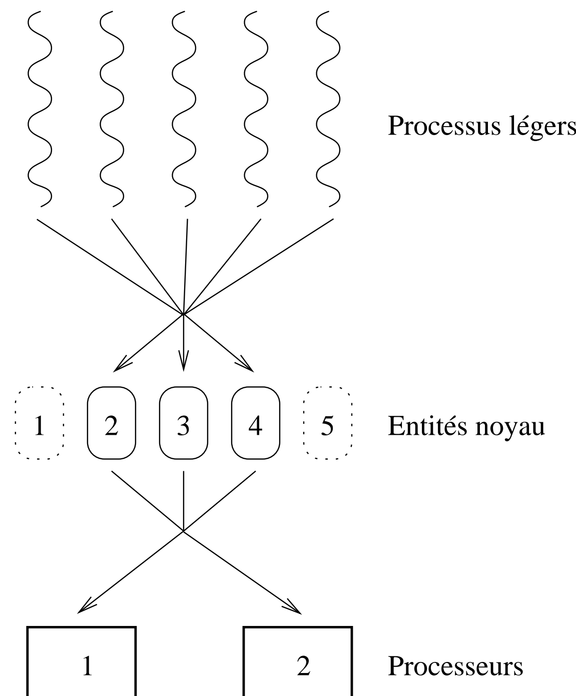

FIG. 3.5 – *Un ensemble de fils d'exécution se partage un ensemble d'entités noyau.*

Quand une entité noyau se bloque en attente d'une entrée sortie ou autre, le noyau en informe la bibliothèque de niveau utilisateur afin que celle-ci engendre un autre processus léger pour maintenir au niveau souhaité le nombre de processus légers en cours d'exécution. Le noyau peut également autoriser le programmeur à modifier le nombre d'entités noyau disponibles.

Cette façon de faire présente des avantages très nets. Elle permet d'exploiter pleinement les machines à multi-processeurs, offre de bonnes performances puisqu'une grande partie des commutations de contexte entre les processus légers se fait en niveau utilisateur, est facilement extensible (*scalable*) puisque les ressources noyau utilisées sur chaque processeur sont limitées. De plus, la latence quand un processus se bloque dans le noyau est faible, puisqu'une autre entité système est immédiatement réactivée.

Cette implantation ne présente pas d'inconvénients particuliers, si ce n'est la complexité de sa mise en œuvre. Il faut en effet gérer non seulement la création de nouvelles entités noyau, mais la destructions d'entités noyau quand celles-ci deviennent inutiles. Cet aspect peut être laissé à la charge de l'application ou géré par le système, qui peut par exemple détruire les entités noyau n'ayant pas servi depuis "un certain temps". Il faut prendre garde aux phénomènes d'instabilité où des entités noyaux sont sans arrêt détruites puis recréées.

3.6 Ordonnancement

Une politique d'ordonnacement a pour rôle de fixer l'ordre d'exécution des fils quand il y a d'avantage de fils prêts à s'exécuter que de processeurs disponibles pour leur exécution [33, 62]. Le choix se ramène à décider du moment où un fil cesse son exécution et du prochain fil à exécuter sur chaque processeur. On distingue les politiques **préemptives**, c'est à dire autorisant la "perte" du processeur à tout moment par un fil au profit d'un autre, de celles **non-préemptives**, quand pour "perdre" le processeur, un fil doit s'être au préalable explicitement bloqué. On distingue également les ordonnancements selon qu'ils offrent ou non la notion de priorité. Celle-ci permet de définir l'importance relative des fils d'exécution pour effectuer un partage non équitable de la ressource processeur. Les ordonnanceurs gèrent habituellement une file d'attente des fils prêts à s'exécuter (cas sans priorité) ou une file d'attente des fils prêts par niveau de priorité ("prêt" s'entend au sens de la figure 3.2, page 32).

3.6.1 Ordonnancement FIFO

Dans l'ordonnancement FIFO (*first in, first out*) sans priorités, chaque fil s'exécute tant qu'il n'a pas terminé ou n'est pas bloqué. Ensuite, le fil prêt qui attend depuis le plus longtemps s'exécute jusqu'à bloquer ou terminer etc… L'ordonnancement **FIFO sans priorités** est un ordonnancement **non-préemptif**.

Le comportement de l'ordonnacement FIFO avec priorités est différent, dans la mesure où un fil s'exécute jusqu'à bloquer ou terminer, ou jusqu'à ce qu'un fil dont la priorité est supérieure (à celle du fil en exécution) passe dans l'état prêt. Si tel est le cas, le fil en cours d'exécution est arrêté et le fil de priorité supérieure s'exécute. Aucun fil ne peut s'exécuter tant qu'il existe des fils prêts dont la priorité est supérieure. L'ordonnancement **FIFO avec priorités** est un ordonnancement **préemptif**. (Dans un ordonnancement FIFO avec priorités mais sans préemption, dans lequel les priorités

ne sont examinées pour déterminer le prochain fil à exécuter que quand le fil en exécution se bloque, un fil de basse priorité peut empêcher un fil de plus haute priorité de s'exécuter pendant un délai non borné).

Selon les implantations, il est possible de demander un ordonnancement relatif aux autres fils d'exécution du même processus lourd ou encore un ordonnancement relatif à tous les fils d'exécution qui s'exécutent sur le même processeur. Dans le cas d'un ordonnancement par rapport à tous les fils d'exécution du système, un fil qui ne se bloque pas et ne termine pas peut paralyser tout le système.[5] Quand l'ordonnancement est fait uniquement à l'intérieur d'un processus lourd, un tel fil bloquera uniquement les autres fils de ce processus lourd, le reste du système continuant à s'exécuter normalement.

3.6.2 Ordonnancement *round-robin*

L'ordonnancement *round-robin* (qui peut être traduit par "à tour de rôle") est similaire à l'ordonnancement FIFO à ceci près qu'un fil est aussi préempté après s'être exécuté sans interruption pendant un certain quantum de temps (défini par l'implantation). Comme dans l'ordonnancement FIFO, il n'est pas possible pour un fil de s'exécuter si un fil de priorité supérieure est prêt. Un ordonnancement *round-robin*, **avec ou sans priorités**, est toujours un ordonnancement **préemptif**.

3.6.3 Autres ordonnancements

En plus des ordonnancements précédents, on en trouve souvent d'autres dans les implantations POSIX de processus légers. Le standard ne les définit pas, mais il n'est pas rare d'en trouver de type "temps partagé", dans lequel tous les fils d'exécution s'exécutent à tour de rôle, ceux de basse priorité s'exécutant même si des fils de plus haute priorité sont prêts. Chaque fil qui s'exécute est préempté après un certain quantum de temps, ce quantum étant fonction de la priorité du fil : les fils de haute priorité s'exécutent plus de temps que les fils de basse priorité.

3.6.4 Priorités et *mutex*

Dans les ordonnancements à priorités, il se peut qu'un fil de basse priorité empêche un fil de plus haute priorité de s'exécuter [11, 62]. Un exemple simple (sur monoprocesseur) est celui d'un fil de basse priorité ayant verrouillé un *mutex* qu'un fil de haute priorité désire verrouiller. Le fil de haute priorité est bloqué pendant la durée de la section critique du fil de basse priorité. Ceci s'appelle "inversion de priorité bornée" (*bounded priority inversion*) puisqu'après la durée de la section critique du fil de basse priorité, les choses entrent dans l'ordre et le fil de haute priorité peut s'exécuter [55].

5. Dans ce cas, il faut un privilège spécial (super-utilisateur) pour pouvoir utiliser l'ordonnancement FIFO.

Des configurations plus gênantes peuvent se présenter (figure 3.6). Supposons les conditions du cas précédent, avec un fil de priorité moyenne qui est réveillé pendant l'exécution de la section critique du fil de basse priorité (section critique protégée par le *mutex* M). Le fil de basse priorité est préempté pendant tout le temps où le fil de priorité moyenne s'exécute. La conséquence est que le fil de priorité moyenne empêche le fil de haute priorité (qui attend la libération du *mutex*) de s'exécuter, et ceci peut durer longtemps (l'attente n'est pas bornée : *unbounded priority inversion*).

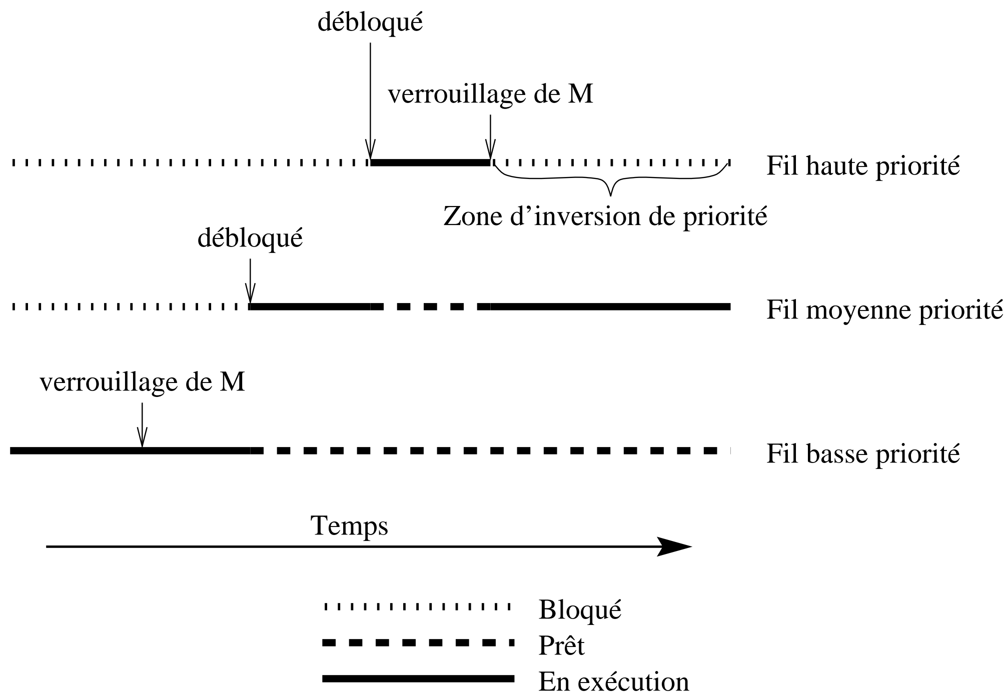

FIG. 3.6 – *Inversion de priorité non bornée sur un monoprocesseur.*

Deux solutions sont possibles pour ce problème et consistent à changer la priorité d'un fil ayant verrouillé un *mutex*.

La première solution, "protocole de priorité plafond" (*priority ceiling protocol*), consiste à associer à chaque *mutex* une valeur appelée priorité plafond. Chaque fois qu'un fil verrouille le *mutex*, sa priorité est augmentée jusqu'à cette valeur, si elle n'y est pas déjà supérieure. Avec le choix d'une valeur de priorité plafond suffisamment grande, le cas d'inversion de priorité ne peut plus se produire. Ce protocole est facile à implanter, la difficulté consistant à trouver les bonnes valeurs des priorités plafond.

La deuxième solution, "héritage de priorité" (*priority inheritance*), est plus complexe mais d'un emploi plus souple. Lors du verrouillage d'un *mutex* le fil conserve sa priorité. Si avant qu'il ne déverrouille le *mutex*, d'autres fils tentent de le verrouiller et se bloquent, le fil possédant le *mutex* voit sa priorité augmentée à la plus grande des valeurs de priorité des fils en attente. De cette façon, ce fil ne peut pas être préempté par des fils de priorité plus basse que les fils en attente et le phénomène d'inversion de

priorité non bornée ne peut se produire.

3.7 Processus légers et codes existants

Des problèmes se posent quand une application multiprogrammée utilise des bibliothèques non prévues pour un tel fonctionnement. Les fonctions de la bibliothèque peuvent ne pas fonctionner correctement en présence de multiples fils d'exécution [11, 62]. Ce problème est lié au fait que la multiprogrammation légère est relativement "jeune", et les anciennes bibliothèques ne sont pas prévues pour un tel fonctionnement.

3.7.1 Compatibilité d'une fonction avec la multiprogrammation légère

Une fonction ou une bibliothèque peuvent être plus ou moins adaptées à la multiprogrammation. Les définitions suivantes peuvent s'appliquer à une fonction isolée ou à une bibliothèque. Dans ce dernier cas, la notion d'appels concurrents s'entend quand ces appels se font à une même fonction ou à des fonctions différentes de la bibliothèque.

Une bibliothèque ou une fonction sont dites **réentrantes** quand plusieurs exécutions concurrentes peuvent y avoir lieu. Il existe deux types de réentrance : celle relative à de multiples fils d'exécution (*thread safe* et *thread aware*) et celle relative au traitement des signaux (*async-safe*). La non réentrance est appelée *thread unsafe*.

Thread unsafe. Un seul fil d'exécution peut appeler la fonction ou bibliothèque sous peine de comportement erroné. Il se peut également que seul le fil initial d'un processus multiprogrammé soit autorisé à effectuer les appels, ce qui équivaut du point de vue de la fonction à un processus monoprogrammé.

Thread safe. La fonction ou la bibliothèque ont un comportement correct lorsqu'elles sont appelées par de multiples fils d'exécution concurremment, mais dans le cas où un des appels est bloquant c'est le processus lourd en entier qui est bloqué et pas uniquement le fil ayant effectué l'appel.

Thread aware. La fonction ou bibliothèque fonctionnent correctement lorsqu'elles sont appelées par de multiples fils d'exécution. En cas de blocage, seul le fil ayant effectué l'appel est bloqué, les autres continuant leurs exécutions.

Async-safe. Cette définition s'applique à une ou des fonctions particulières d'une bibliothèque, mais rarement à la bibliothèque dans son ensemble. Une fonction *async-safe* est une fonction pouvant être appelée à partir d'un traitant de signal (*signal handler*). Plus précisément, le comportement d'un système dans lequel une fonction *async-unsafe* est appelée par un traitant de signal ayant interrompu une fonction *async-unsafe* est indéterminé.

3.7.2 Causes possibles des problèmes

Il existe de nombreuses raisons pour lesquelles une fonction ou une bibliothèque peuvent ne pas fonctionner correctement en présence de multiples fils d'exécutions [61] :

1. Des données privées à la fonction ou à la bibliothèque présentent des états incohérents quand elles sont accédées concurremment par de multiples fils d'exécution.

2. Une fonction maintient dans ses variables privées un état entre deux appels successifs. Les états relatifs à différents fils d'exécution vont se mélanger quand les appels de ces fils s'entrelacent.

3. Utilisation de variables allouées statiquement pour stocker des résultats intermédiaires lors d'un calcul et plus généralement implantations non réentrantes de fonctions dont l'interface est réentrante.

4. Une fonction retourne un pointeur vers des données statiques pouvant être modifiées par un fil alors qu'un autre fil est en train de les utiliser et plus généralement fonction dont l'interface (profil de la fonction) est non réentrante.

5. Lors d'un appel d'une fonction bloquante, tous les processus légers se bloquent, et pas uniquement l'originaire de l'appel.

6. La bibliothèque ne gère pas correctement l'annulation (*cancellation*) des processus légers.

Si les points 1 à 4 concernent la façon dont la bibliothèque ou la fonction sont écrites, indépendamment des processus légers, les points 5 et 6 sont relatifs à l'intéraction proprement dite avec les processus légers. Le point 5 est même complètement dépendant de la bibliothèque de processus légers utilisée et de la façon dont les fonctions des bibliothèques standard ont été adaptées à la multiprogrammation légère.

Des solutions existent à certains de ces problèmes, même si elles ne permettent pas toujours de transformer une bibliothèque *thread unsafe* en bibliothèque *thread aware*. Parmi ces solutions on trouve :

– Interdiction par verrou à plusieurs fils de s'exécuter concurremment dans la bibliothèque. Ce verrouillage peut être interne (dans les fonctions de la bibliothèque) ou externe (l'appelant se charge de sérialiser ses appels). Cette technique peut permettre de résoudre les problèmes des cas 1 et 3. Pour le cas 4 les verrous peuvent être également utiles sous condition que le fil maintienne le verrou pendant toute la durée de l'accès au résultat de la fonction. Les deux types de verrouillages interne et externe ne sont pas exclusifs.

– Utilisation de zones de données privées. Ces zones peuvent permettre de gérer l'allocation des données privées des cas 1, 2 et 3 et d'utiliser des zones mémoire différentes pour les résultats du cas 4.

– Utilisation uniquement de fonctions non-bloquantes de la bibliothèque. Ceci tente de résoudre le problème posé par le cas 5. Les fonctionnalités bloquantes sont simulées à l'aide d'appels non bloquants effectués périodiquement et de synchronisations entre processus légers.

– Utilisation de mécanismes de nettoyage lors de la disparition des fils, afin de libérer les ressources relatives à la bibliothèque que ces fils peuvent posséder (cas 6).

Les "solutions" énumérées ci-dessus ne sont pas toujours applicables (selon que l'on dispose du code source de la bibliothèque à modifier etc...), et ne suffisent pas toujours pour atteindre l'objectif d'une bibliothèque *thread aware*. Il est parfois plus aisé de réécrire certaines fonctions plutôt que d'essayer de rendre réentrantes des fonctions dont l'interface ne l'est manifestement pas.

3.8 Bilan

Les processus légers, après avoir été pendant longtemps un sujet de recherche, sont aujourd'hui intégrés dans un nombre croissant de systèmes, d'applications et de langages. La standardisation dont ils ont fait l'objet facilite l'écriture de programmes portables et la plupart des grands fabriquants d'ordinateurs proposent les *Pthreads* sur leurs machines. Les fonctionnalités de ce standard couvrent les besoins d'une large classe d'applications parallèles, distribuées ou "classiques". Ces noyaux ont l'avantage d'être disponibles facilement, et leur grande diffusion fait qu'ils ont été débarrassés d'un grand nombre de bogues.

À coté des ces noyaux standards en existent d'autres, qui en reprennent en partie les fonctionnalités et qui en proposent d'autres plus "exotiques". Par exemple le noyau Marcel utilisé par PM2 (voir [73]) fournit des mécanismes d'extension de pile, d'hibernation et de réveil.

L'utilisation des processus légers s'avère intéressante pour masquer des délais d'attente lors de l'exécution (intéraction avec l'utilisateur, entrées sorties...) et c'est donc naturellement que la multiprogrammation légère a été utilisée dans les environnements de communication. Le chapitre suivant présente ce sujet.

Chapitre 4

Multiprogrammation et communications

4.1 Introduction

L'intégration des communications et de la multiprogrammation légère présente de nombreux avantages, liés tant à la recherche de l'efficacité dans l'exécution qu'à une plus grande souplesse et facilité de programmation. Parmi ces avantages on peut noter :

– Les processus légers ont un surcoût faible comparé à celui des processus lourds. Ils permettent l'exécution efficace de tâches à grain fin au sein d'une application parallèle ou distribuée.

– L'ordonnancement des processus légers peut être transparent au programmeur et "automatique" (on parle d'"auto-ordonnancement"). Il permet d'ignorer l'indéterminisme associé aux communications, dû à la durée non connue a priori d'une communication.

– La multiprogrammation permet de recouvrir les attentes de communication d'un fil par des calculs d'un autre fil, masquant de ce fait les durées de communication.

– L'écriture d'une application à l'aide de plusieurs fils d'exécution peut être faite pour des raisons de simplification de la programmation et de la maintenance. Une intégration des communications et de la multiprogrammation légère permet d'étendre cette démarche aux applications communiquantes. En général, les raisons évoquées section 3.3, page 29 et suivantes, s'étendent aux applications parallèles et communicantes.

Quand des processus légers communiquent, se pose la question du nommage des destinataires des messages. Dans le modèle processus communiquants traditionnel,

des identificateurs sont associés aux processus, sont connus sur l'ensemble de la machine parallèle et servent d'adresses d'origine et de destination des messages. Dans une machine parallèle composée de machines indépendantes reliées par un réseau, de tels identificateurs sont par exemple composés du nom de la machine concaténé au numéro de processus sur la machine. Une telle démarche est délicate à étendre au cas des communications en présence de processus légers, car ceux-ci, contrairement aux processus lourds, ne sont pas des entités "stables". En effet, le coût de création des processus lourds fait qu'ils ont une longue durée de vie, car la performance d'une application ayant recours à une création dynamique systématique serait fortement dégradée. Par contre, dans une application utilisant la multiprogrammation légère, il est courant que des fils soient créés pour des tâches ponctuelles et courtes et que ces fils aient néanmoins besoin de communiquer. Un mécanisme de nommage dans lequel chaque processus léger se voit attribuer un nom global connu sur l'ensemble de la machine peut d'une part induire un surcoût important et d'autre part être peu pratique à l'emploi.

Une alternative à l'envoi des messages à un fil particulier consiste à envoyer les messages à une boîte aux lettres, qui est ensuite lue par un ou plusieurs fils. Cette approche de communication anonyme présente de nombreux avantages car elle est d'implantation plus simple, d'usage plus général et permet qui plus est de reconstruire le cas de l'adressage des messages à un fil particulier. L'implantation est plus simple car elle ne nécessite pas de tenir compte de la création et de la disparition des fils d'exécution pour la gestion des messages. Elle est d'usage plus général car un fil peut communiquer sur plusieurs boîtes aux lettres simultanément et une boîte peut également servir plusieurs fils. Enfin, pour reconstruire les communications de fil à fil il suffit d'associer à chaque fil une boîte aux lettres vers laquelle seront envoyés les messages qu'il sera seul à lire.

4.2 Définitions

Les supports de communication intégrant la multiprogrammation légère reprennent une grande partie sinon la totalité des concepts des bibliothèques de communication traditionnelles. Les termes définis en section 2.2, page 16 sont également utilisés dans ce chapitre. D'autres termes sont néanmoins introduits du fait des nouvelles fonctionnalités de ces supports.

Tâche : contrairement aux bibliothèques classiques de communication, dans lesquelles la tâche est l'entité de base de la machine parallèle, dans les bibliothèques intégrant communications et multiprogrammation légère la tâche est le conteneur des entités de base de l'exécution, à savoir les processus légers.

nœud : dans le cas d'une machine monoprocesseur, ce terme désigne un processeur unique. Dans le cas d'une machine multi-processeur à mémoire commune, le terme "nœud" peut désigner soit un processeur, soit l'**ensemble** des processeurs

de la machine, selon le contexte. Pour les bibliothèques de communication classiques (sans multiprogrammation légère) cette double définition n'était pas nécessaire car un processus lourd ne pouvait s'exécuter que sur un processeur physique à la fois, ce qui n'est plus le cas en présence de processus légers.

Service : fonction s'exécutant dans une tâche, la plupart du temps sous la forme d'un nouveau processus léger créé pour l'occasion, à la demande d'une autre tâche (ou d'un processus léger d'une autre tâche). Un service accepte des paramètres à sa création mais ne renvoie pas de résultat.

Service urgent : service exécuté sans la création d'un nouveau processus léger dans la tâche, mais par un processus léger préexistant. Des contraintes de non blocage s'appliquent aux services urgents. Dans certains systèmes la distinction entre service et service urgent est nette, dans d'autres la frontière est plus floue (un service urgent pouvant par exemple se transformer en service normal en cas de blocage).

Appel de procédure à distance (RPC) : dans le contexte des environnements de communication et de multiprogrammation légère, l'appel de procédure à distance est un appel "léger", dans lequel la procédure est exécutée par un processus léger et non pas par un processus lourd comme pour l'appel classique de procédure à distance. Contrairement à l'appel de service, l'appel de procédure à distance renvoie un résultat.

Point d'entrée : lieu du début de l'exécution d'un service, d'une procédure appelable à distance ou d'un processus léger. Par exemple le nom de la fonction exécutée.

Déclaration de service : un service ne s'exécute le plus souvent pas dans la tâche l'invoquant, car la requête de service peut être distante. Les espaces d'adressage des tâches sont disjoints, et même si les codes exécutés sont identiques, rien ne garantit qu'une même fonction aura la même adresse dans deux tâches différentes. Les déclarations sont faites pour que les tâches puissent s'entendre sur un nommage global cohérent des services.

4.3 Systèmes existants

4.3.1 TPVM

TPVM [34] est une extension de PVM aux processus légers, développée sans modifier le code de PVM et utilisant divers noyaux de multiprogrammation légère. TPVM se présente comme une application PVM, et de ce fait bénéficie de l'infrastructure de PVM concernant l'installation de la machine parallèle et son évolution dynamique (variation du nombre de nœuds, de tâches. . .).

TPVM possède deux modèles de programmation. Le premier, classique, de processus légers communiquant par échange de messages, et le second, *dataflow* gros-grain, dans lequel chaque exécution d'un processus léger dépend de la disponibilité d'un ensemble de messages et est déclenchée quand ces messages arrivent. Dans les deux cas, TPVM présente une implantation "légère" du modèle classique des processus communiquants et interdit de ce fait à des fils résidant dans une même tâche de communiquer par mémoire commune ou de se synchroniser par les primitives habituelles.

Dans les deux modèles, la création d'un processus léger à distance suppose la déclaration préalable du point d'entrée correspondant. Dans le modèle *dataflow* gros-grain sont également déclarées les dépendances du point d'entrée, à savoir les étiquettes des messages devant être disponibles avant la création effective d'un processus léger exécutant ce point d'entrée.

Dans le modèle classique, la création d'un fil se fait en donnant le nom de son point d'entrée, le nombre de copies (de fils différents) à créer et le numéro de la tâche PVM sur laquelle le fil doit être créé (ce dernier paramètre peut être laissé au choix du système). L'appel bloque jusqu'à ce que les fils aient été créés et retourne alors un tableau contenant leurs identificateurs (l'identificateur d'un fil contient l'identificateur de la tâche qui l'héberge). Dans le modèle *dataflow* gros-grain il n'y a pas de création explicite de fils : quand l'ensemble des messages dont dépend un fil a été invoqué (terme utilisée dans la littérature TPVM pour désigner les "envois" du modèle *dataflow* gros-grain), ce fil sera automatiquement créé sur le processeur le mieux adapté en fonction de la charge du système et pourra alors recevoir les messages. La synchronisation des fils est donc implicite, basée sur la dépendance des données. Les deux modèles de programmation peuvent être utilisés conjointement au sein d'une même application.

L'échange de données entre les fils, lors d'une création explicite ou suite à la satisfaction d'un ensemble de dépendances, se fait à l'aide de tampons envoyés dans des messages. Le méthode est similaire à celle de PVM dans laquelle les données sont emballées dans le tampon coté expéditeur et déballées coté récepteur. Les messages sont explicitement envoyés de fil à fil avec adressage explicite.

L'implantation de TPVM fait appel à un fil particulier par machine parallèle, le fil serveur, qui gère la base de données globale des exportations des points d'entrée, avec leurs dépendances le cas échéant ainsi que les données concernant les invocations en cours. De plus, sur chaque tâche se trouve un autre fil particulier, le fil principal, interagissant avec le fil serveur et gérant localement les exportations et les invocations en cours. Lors d'une réception TPVM, un fil essaye de recevoir directement les messages qui lui sont adressés à l'aide d'un appel PVM. Si à cette occasion il reçoit un message destiné à un autre fil, il le range dans une file prévue à cet effet. Si aucun message n'est disponible, il cède la main à un autre fil. Si tous les fils d'une tâche sont en attente de messages, la tâche cède la main à un autre processus Unix.

L'implantation de TPVM est faite au dessus de la bibliothèque PVM non modifiée et de divers noyaux de processus légers. TPVM a été développée sur deux types de noyaux de processus légers : une bibliothèque de processus légers non préemptifs en espace utilisateur (Rex [63, 20]), et une bibliothèque de processus légers préemptifs,

avec support dans le noyau (SunOS 5.3). Dans le deuxième cas, des primitives de synchronisation ont été utilisées pour réguler l'ordonnancement des processus légers.

4.3.2 Nexus

Nexus [40, 39, 41] est un exécutif parallèle destiné à être une cible pour compilateurs. De nombreux logiciels s'en servent comme couche de communication et de multiprogrammation, comme Compositionnal C++, Fortran M, nPerl, MPICH (dont Nexus est une des bibliothèques sur lesquelles peut être développée l'ADI, *Abstract Device Interface*), CAVEcomm etc... Nexus est destiné à constituer un support efficace d'exécution en milieu hétérogène, tant au niveau des machines qu'au niveau des réseaux. Nexus peut être développé au dessus de bibliothèques standard d'échange de messages et de multiprogrammation. Des bibliothèques différentes de multiprogrammation peuvent être utilisées sur des machines différentes participant à un même calcul. Ceci constitue une souplesse supplémentaire en milieu hétérogène, quand une même bibliothèque n'est pas disponible sur toutes les machines. De même, plusieurs protocoles réseau différents peuvent être utilisés dans un même programme.

Une machine parallèle Nexus est vue comme un ensemble de nœuds pouvant évoluer dynamiquement, sur lesquels sont placés des tâches (appelées "contextes" dans la terminologie Nexus), avec dans chaque tâche des processus légers qui s'exécutent. Plusieurs tâches peuvent être placées sur chaque nœud et à sa création, une tâche est vide de tout processus léger (sauf la tâche initiale de la machine parallèle dans laquelle s'exécute un processus léger dès sa création). La création d'une tâche renvoie un pointeur global vers cette tâche (voir paragraphe suivant), ceci afin de permettre d'y créer ultérieurement des fils d'exécution.

Il est possible à un fil de créer localement d'autres fils dans sa tâche. Les fils d'une même tâche peuvent se synchroniser par les primitives usuelles.

Nexus introduit les notions de pointeur global (*Global Pointer*: GP) et d'appel de service à distance (*Remote Service Request*: RSR). Un pointeur global, comme son nom l'indique, est un nom global à l'ensemble de la machine, désignant une adresse particulière dans une tâche donnée. Les pointeurs globaux permettent entre autres choses la création de structures chaînées embrassant l'ensemble de la machine parallèle. Un pointeur global contient également les protocoles de communication supportés par la tâche vers laquelle il pointe (ou plus précisément par le nœud de la tâche vers laquelle il pointe). Ainsi Nexus peut déterminer quel est le protocole le plus adapté à un RSR.

L'appel de service à distance est la seule primitive de communication offerte par Nexus et le seul moyen de créer des fils d'exécution à distance. Quand un fil effectue un tel appel, il passe un pointeur global, un identificateur de service et un tampon de paramètres. Le service s'exécutera dans la tâche vers laquelle pointe le pointeur global et recevra en paramètre le tampon ainsi que l'adresse locale extraite du pointeur global. L'exécution pourra se faire soit par un processus léger créé pour l'occasion, soit par un processus léger système préexistant. Dans ce dernier cas certaines contraintes de non

blocage s'appliqueront au service mais en contrepartie l'appel sera plus rapide.

Le RSR est une opération asynchrone pour le fil qui l'initie. Il ne se bloque pas comme dans un appel de procédure à distance (RPC) car il n'y a pas de retour d'une telle opération. Une fonctionnalité de type RPC doit être construite à l'aide de deux RSRs : un pour l'appel et l'autre pour le retour.

Nexus propose plusieurs implantations pour les RSR, en fonction des propriétés du noyau de processus légers utilisé et des fonctionnalités de la machine :

– Quand le noyau de processus légers ne permet pas de bloquer un seul fil d'exécution sur une communication (cas des implantations en espace utilisateur), Nexus propose d'affecter un fil spécialisé, le *communication thread*, à la scrutation du réseau pour le compte des autres fils. Il est également possible dans ce cas de faire effectuer la scrutation par chaque fil qui se bloque sur une communication.

– Quand le système permet aux processus légers d'effectuer des appels bloquants sans paralyser le processus lourd dans son ensemble, un fil en attente de RSR peut se bloquer en réception. Le délai de prise en compte d'un RSR dépendra du type d'ordonnancement du noyau de processus légers.

– L'arrivée d'un RSR peut enfin être signalé par une interruption quand le système le permet. Cette solution peut s'avérer coûteuse du fait du temps mis pour délivrer une interruption à l'application.

Nexus est enfin construit de façon à ne pas imposer des copies supplémentaires des messages, en plus de celles éventuellement faites par la bibliothèque de communication utilisée. Bien sur, cet objectif ne peut être atteint en toutes circonstances. Nexus affiche de bonnes performances [41], avec un surcoût de seulement 80% par rapport à la bibliothèque support (MPL) pour un échange de messages de taille nulle (temps de latence) sur une machine IBM SP2. Étant donnée l'efficacité de MPL, un tel surcoût n'est pas très élevé.

4.3.3 Chant

Chant [52, 53] intègre communication et multiprogrammation légère au dessus de bibliothèques standard et offre les communications point à point entre deux processus légers ainsi que les appels de service à distance.

Les processus légers sont identifiés par un triplet comprenant le groupe, le rang de la tâche dans le groupe et le rang du processus léger dans la tâche. Ce type de nommage est similaire à celui employé par MPI et offre l'envoi de messages de fil à fil avec adressage explicite. Dans une même tâche les fils d'exécution peuvent communiquer par mémoire partagée et par les mécanismes habituels. Les fonctionnalités du noyau de processus légers peuvent être directement utilisées localement, et sont étendues pour s'appliquer à distance (par exemple un `join` sur un processus léger distant).

Par souci d'efficacité Chant n'impose pas de copies inutiles des messages, l'identité du fil destinataire est donc codée dans l'étiquette du message (si l'identité était passée dans le corps du message, il faudrait effectuer une copie du message à l'émission pour l'insérer et une copie à la réception pour l'extraire). Ceci interdit l'utilisation de jokers (*wildcards*) dans une réception.

Comme dans tous les autres systèmes, les services doivent être déclarés au système. Lors de la déclaration il est nécessaire de préciser si le service risque de bloquer, afin de l'exécuter le cas échéant dans un fil d'exécution séparé. L'appel de service à distance est pris en charge par un fil spécialisé.

Afin de gérer les communications des différents fils et les appels de service à distance, Chant effectue des scrutations du réseau. Plusieurs possibilités sont proposées et évaluées :

– Scrutation par le fil effectuant la communication. Cette solution économise l'enregistrement de la communication avec un fil spécialisé ou avec le système mais génère beaucoup de commutations de contexte.

– Scrutation par l'ordonnanceur des processus légers. À chaque ordonnancement on peut tester les communications bloquées de tous les fils d'exécution ou encore uniquement celle du fil devant être ordonnancé. Cette solution implique de modifier l'ordonnanceur des processus légers, ce qui n'est pas toujours possible, et peut également générer trop de scrutations. Elle évite par contre la double commutation de contexte ayant lieu quand un fil scrute le réseau et reste bloqué.

– Scrutation par un fil spécialisé. Cette solution évite la modification de l'ordonnanceur et une scrutation trop fréquente. Il faut par contre s'assurer que ce fil spécialisé sera exécuté assez souvent. Quand beaucoup de communications sont bloquées, cette solution peut être plus coûteuse que la scrutation par chaque fil car elle a un grand nombre de communications à vérifier.

Quand le système permet de tester toutes les communications bloquées en même temps, les deux dernières possibilités gagnent en efficacité. Il est rapporté un surcoût faible (de l'ordre de 10%) quand chaque fil effectue sa propre scrutation par rapport à la modification de l'ordonnanceur, la scrutation par fil spécialisé donnant des résultats moins bons.

Chant a été développé et testé sur stations Sun, en utilisant p4 [12] et pthreads[1] [71, 72] et sur la machine Intel Paragon utilisant NX [79] et un petit noyau de processus légers développé pour l'occasion. Il est portable vers toute architecture supportant MPI et les *Pthreads*. Chant exploite la préemption et les priorités du noyau de processus légers pour borner le délai entre deux opérations de scrutation.

1. Ne pas confondre "pthreads", qui est le nom d'un développement spécifique d'un noyau de processus légers (voir les références bibliographiques) et *Pthreads*, qui est l'abréviation de "POSIX *threads*".

4.3.4 MPI-F

MPI-F [44, 42, 45] est un prototype développé par IBM pour les machines SP1 et SP2, présentant l'interface standard MPI et intégrant des processus légers[2]. MPI-F utilise du code provenant de MPICH pour les opération de haut niveau (communications collectives, communicateurs) mais est complètement réécrit au dessus des couches basses de communication (*packet layer* et *pipe-layer* utilisées également par MPL, la bibliothèque de communication "native" des SP1/SP2). Le noyau de multiprogrammation légère DCE a été "allégé" puis complètement intégré à cette version *thread-aware* de MPI.

Pour une efficacité maximale, MPI-F utilise pour ses communications l'adaptateur de communication HPS du SP1 "*mappé*" dans l'espace d'adressage de la tâche (processus utilisateur) : les opérations de communication se font depuis l'espace d'adressage utilisateur. Cette approche évite les copies inutiles et les appels système coûteux mais implique l'utilisation exclusive de l'adaptateur de communication sur le nœud (une seule tâche par nœud). L'avancement des communications se fait à chaque appel à MPI-F mais également à l'aide d'une horloge, pour les cas où aucun fil n'appelle MPI-F pendant un long moment.

En plus de l'interface définie par MPI, MPI-F offre l'appel de service à distance (RSR : *Remote Service Request*) sous trois variantes : *basic*, *threaded* et *quick*. Les trois formes permettent de faire l'appel et de passer des paramètres, mais pas de recevoir un résultat. La forme *basic* exécute le service distant dans un fil système dédié à cet effet, qui ne peut exécuter qu'un service à la fois. Dans la version *threaded* un nouveau fil d'exécution est créé sur le nœud distant pour exécuter le service et dans la version *quick* c'est le fil gérant la réception des RSR qui exécute le service. Différentes contraintes de programmation s'appliquent à chacune des trois formes de RSR.

MPI-F est la version de MPI utilisée pour la première version d'Athapascan-0b. Néanmoins, les fonctionnalités de MPI-F relatives aux RSR et à la multiprogrammation légère n'ont pas été utilisées, par souci de portabilité.

4.3.5 PM2

PM2 (*Parallel Multithreaded Machine*) [73] est un système construit au dessus de PVM et de Marcel[3], un noyau de processus légers préemptifs à ordonnancement de partage de temps.

Le modèle de programmation de PM2 repose sur l'appel de procédure à distance (RPC). Un service, préalablement déclaré, est exécuté sur une tâche distante dans un fil d'exécution créé pour l'occasion ou dans un fil système (RPC classique dans le premier cas et *quick* dans le second dans la terminologie PM2). Trois variantes du RPC existent pour chacun des deux modes : l'appel synchrone classique, dans lequel l'appelant est

---

2. MPI-F est aujourd'hui abandonné car la version "produit" du MPI d'IBM tout comme AIX 4.1 intègrent la multiprogrammation légère.

3. Le nom "Marcel" est un hommage à Proust et à sa recherche du temps perdu...

suspendu tant que le message de retour n'est pas arrivé, l'appel à attente différée dans lequel l'appelant peut continuer son exécution avant d'attendre le message de retour et l'appel asynchrone, dans lequel il n'y a pas de message de retour (il s'agit alors d'un appel de service à distance). Les appels de procédure à distance s'effectuent par des fonctions souches (*stubs*), dont le rôle est de cacher le fait que l'appel se fait à distance en présentant une interface de fonction “normale”. Les fonctions souches emballent les paramètres pour leur envoi sur le réseau, effectuent l'appel distant et déballent les résultats au retour. L'utilisateur doit fournir les primitives d'emballage et de déballage des données, le reste des fonctions souche étant généré par le système.

Les seuls échanges de données entre fils d'exécution se font lors des appels et retours des RPC. Les fils s'exécutant dans une même tâche ne communiquent pas entre eux par mémoire commune.

Le noyau Marcel intègre un support pour la migration de processus légers que PM2 exploite. Un processus léger peut donc migrer de façon “transparente” d'une tâche PVM vers une autre. Cette migration est facilitée par le modèle de programmation qui n'autorise que les RPC, puisque le système gère la localisation des fils appelants pour leur acheminer l'éventuel message de retour (le message d'appel n'est bien sur pas affecté par une migration). Les contraintes de programmation pour qu'un processus léger puisse migrer sont assez fortes et concernent l'utilisation des pointeurs et des zones mémoire allouées dans le tas. Certaines pratiques sont totalement incompatibles avec la migration (utilisation de mémoire allouée dans le tas ou de pointeurs vers des structures globales, d'ailleurs contraire au modèle de programmation de PM2) alors que d'autres le sont temporairement (utilisation de pointeurs locaux) et doivent être protégées en empêchant temporairement la migration durant l'exécution. Un compilateur spécifique (`gcc`) doit également être utilisé.

L'implantation de PM2 se basait au départ sur une version modifiée de PVM, afin de permettre l'utilisation de la préemption et du partage de temps. PM2 a ensuite été modifié afin de pouvoir utiliser les versions optimisées de PVM développées par les constructeurs (dont le source est “rarement” disponible). La différence de performance entre les deux versions étant minime, seule la version utilisant PVM sans en modifier le code est maintenue aujourd'hui.

4.3.6 DTMS

DTMS (*Distributed Tasks Management System*) [19] est un environnement développé au dessus de la couche *sockets* d'Unix et de divers noyaux de processus légers, selon disponibilité sur la machine cible.

Le modèle de programmation comprend l'appel de services (appelés “fonctions comportamentales” dans la terminologie DTMS) à distance, exécutés sous la forme de processus légers (appelés “tâches”[4]). Chaque processus lourd (appelé “module” en

4. Cette terminologie DTMS étant incompatible avec celle adoptée dans ce document, où “tâche” a le sens de “processus lourd”, on n'utilisera pas du tout le terme “tâche” dans cette section, pour éviter

DTMS[5]) déclare les services qu'il accepte d'exécuter et peut également annuler une précédente déclaration. La liste des services déclarés par chaque processus lourd de la machine ainsi que celle des services en cours d'exécution sont présentes sur un serveur responsable du placement des fils d'exécution (il est néanmoins possible de faire un placement explicite). Les fils d'exécution communiquent entre eux par envoi de messages, adressés explicitement de fil à fil. La composition de la machine parallèle peut enfin évoluer dynamiquement par ajout et suppression dynamique de processus lourds et de nœuds ("sites" dans la terminologie DTMS).

L'implantation du serveur recensant les services exportés et ceux en exécution est distribuée, et prend la forme d'un processus lourd sur chaque nœud. Chaque serveur est mis à jour périodiquement (périodicité paramètrable) par échange avec les serveurs des autres nœuds pour avoir une vision globale à jour de la machine.

La gestion des communications des processus légers fait également appel à un processus lourd par nœud. Il s'agit du "démon de communication". Quand un fil d'exécution souhaite envoyer un message, il l'envoie au démon de communication de son nœud qui entre à son tour en communication avec le démon de communication du nœud distant qui transmet enfin le message au fil d'exécution destinataire. L'envoi synchrone génère un accusé de réception qui revient à l'émetteur pour le débloquer. L'envoi asynchrone ne comprend pas cette étape mais étant donné le caractère dynamique des fils d'exécution, il y a un risque de s'adresser à un fil n'existant plus. Les messages sont emballés dans des tampons avant d'être envoyés. Chaque fil ne dispose que d'un tampon qui est reinitialisé automatiquement après chaque envoi (impossibilité d'envoyer le message à plusieurs destinataires sans le reconstruire à chaque fois dans le tampon). DTMS est semblable dans sa structure à PVM à ceci près qu'il offre la multiprogrammation légère.

DTMS est implanté au dessus de Solaris 2.x (appelé aussi SunOS 5.x) et au dessus de OSF/1 et Mach [83]. Ces deux systèmes intègrent le support des processus légers au niveau noyau avec préemption. Il n'y a pas eu de portages de DTMS vers des machines où les fils d'exécution ne sont pas supportés par le système.

4.3.7 Athapascan-0a

Athapascan-0a [17] est une précédente version d'Athapascan développée dans le cadre du projet APACHE, reposant sur PVM non modifié et utilisant divers noyaux de processus légers non préemptifs en espace utilisateur. Athapascan-0a se présente comme une bibliothèque associée à un ensemble de macros structurant la programmation.

Le modèle de programmation d'Athapascan-0a repose sur l'appel léger de procédure à distance vers des points d'entrée exportés au préalable. C'est le seul outil pro-

les confusions.

5. Le concept de "module" de DTMS correspond à ce qui est appelé "tâche" tout au long de cette thèse.

posé pour générer du parallélisme physique et pour communiquer entre tâches. L'appel se fait de façon asynchrone, avec passage de paramètres à l'appel et renvoi de résultats au retour. L'asynchronisme permet de générer du parallélisme, l'appelant continuant à s'exécuter en parallèle avec la procédure distante qui est exécutée dans un processus léger distinct. L'envoi des paramètres et des résultats se fait par leur emballage dans des tampons.

La programmation d'Athapascan-0a ne se fait pas uniquement à l'aide d'appels de fonctions d'une bibliothèque. Athapascan-0a définit un ensemble de macros permettant de structurer la programmation mais l'alourdissant aussi considérablement et empêchant l'utilisation de langages différents de C, tel C++.

Athapascan-0a scrute le réseau temporairement pour découvrir de nouveaux messages. Cette scrutation peut être non bloquante pour partager le processeur entre processus légers ou bloquante pour céder le processeur à d'autres processus lourds quand tous les processus légers sont en attente de communications.

4.4 Résumé

Cette section présente un comparatif des différents environnements décrits ci-dessus, sur les aspects du modèle de programmation et des fonctionnalités associées ainsi que de l'implantation.

4.4.1 Modèle de programmation

On s'intéresse ici aux primitives de génération de parallélisme, c'est à dire générant des activités concourantes physiquement (sur différents processeurs) ou logiquement (sous la forme de multiples fils d'exécution sur un même processeur) et aux moyens qu'ont ensuite ces activités concourantes pour communiquer entre elles.

TPVM. Il existe deux modèles de programmation pouvant être combinés. Dans le premier la création de service à distance se fait par appel de service explicite, dans le deuxième des services sont définis avec un ensemble de dépendances et sont automatiquement activés quand les dépendances sont satisfaites. Les communications se font de façon explicite de fil à fil. Les fils s'exécutant sur une même tâche ne peuvent utiliser les mécanismes habituels de synchronisation.

Nexus. La génération de parallélisme se fait soit par une création locale de fils, soit par un appel de service à distance, avec ou sans création de fil sur le nœud distant. Il n'y a pas de communications point à point à proprement parler, tous les échanges se faisant par appel de services à distance avec passage de paramètres. Sur une même tâche, les fils peuvent utiliser les mécanismes habituels de communication et de synchronisation.

Chant. Le parallélisme s'exprime par la création locale de fils d'exécution et par l'appel de services à distance, avec ou sans création d'un nouveau fil d'exécution. Les communications distantes se font explicitement de fil à fil et localement sur une même tâche les fils bénéficient des primitives habituelles de communication et de synchronisation.

MPI-F. L'appel de service à distance sans résultat, avec création d'un nouveau fil d'exécution ou pas permet de créer du parallélisme. La communication est celle définie par le standard MPI et se fait de tâche à tâche. Une utilisation adaptée des étiquettes des messages permet de communiquer entre fils. Les processus légers d'une même tâche peuvent utiliser les mécanismes habituels de communication et de synchronisation.

PM2. Le parallélisme est créé par l'appel de procédure à distance et l'appel de service à distance. Ces appels entraînent la création d'un fil d'exécution distant. Sous certaines conditions, ces fils peuvent migrer et passer d'un processeur à un autre. Le seul moyen de communication est l'appel de procédure et de service à distance, et les fils d'exécution d'une même tâche ne peuvent communiquer ou se synchroniser.

DTMS. Le parallélisme est créé par l'appel de service à distance qui crée un fil sur le site distant. Les interactions entre fils se font à l'aide de communications explicites, adressées à destination d'un fil particulier. Les fils d'exécutant sur une même tâche ne peuvent pas bénéficier des mécanismes habituels de synchronisation entre processus légers.

Athapascan-0a. La seule opération de création de parallélisme et de communication est l'appel de procédure à distance donnant lieu à la création d'un fil sur une tâche distante. Il n'existe pas de primitives de communication ou de synchronisation explicites y compris pour les fils d'une même tâche.

4.4.2 Implantation

TPVM. L'implantation se base sur divers noyaux de processus légers et la bibliothèque PVM non modifiée. Portabilité aisée sur les plates-formes disposant de PVM.

Nexus. L'implantation peut être faite au dessus d'un grand nombre de bibliothèques d'échange de messages et de noyaux de processus légers. La portabilité est aisée sur la plupart des plates-formes existantes.

Chant. L'implantation a été faite au dessus de diverses bibliothèques d'échange de messages et divers noyaux de processus légers. La portabilité est aisée vers les plates-formes disposant de MPI et de processus légers au standard *Pthreads*.

MPI-F. L'implantation est optimisée pour la machine cible (IBM SP1/SP2) et a été faite au dessus des basses couches de communication à haut débit de la machine, en intégrant une version modifiée d'un noyau de processus légers au plus bas niveau dans la gestion des communications. MPI-F n'est de ce fait absolument pas portable.

PM2. L'implantation repose sur la bibliothèque PVM, utilisée telle qu'elle sans modifications, et un noyau de processus légers "Marcel" développé par l'équipe de PM2. Le portage de PM2 implique la disponibilité de PVM et le portage (aisé) de Marcel.

DTMS. L'implantation est faite au dessus de la couche *sockets* en utilisant le support des processus légers offert par le système. DTMS est facilement portable sur les architectures supportant les processus légers au niveau système.

Athapascan-0a. L'implantation utilise divers noyaux de processus légers non préemptifs et la bibliothèque PVM, non modifiée. La portabilité est aisée sur les plates-formes disposant de PVM.

4.5 Bilan

Il peut être intéressant de situer Athapascan-0b par rapport aux environnements décrits ci dessus. La philosophie derrière le modèle de programmation d'Athapascan-0b est de ne pas imposer de contraintes "inutiles" au programmeur, à savoir des contraintes non nécessaires à l'intégration des communications et de la multiprogrammation légère. C'est ainsi qu'Athapascan-0b permet entre autres de créer des processus légers localement et à distance, de les synchroniser et de les faire communiquer. Au niveau de l'implantation, le choix a été celui de la simplicité par l'utilisation de bibliothèques standard (MPI et les processus légers POSIX). Le chapitre suivant donne une description détaillée d'Athapascan-0b.

On distingue deux catégories d'environnements combinant communications et multiprogrammation légère. Dans la première, les fonctionnalités de communication et celles de multiprogrammation légère sont intégrées de façon "légère", laissant clairement apparaître dans l'interface de l'environnement les deux ensembles sous-jacents : le programmeur dispose des fonctionnalités classiques des processus légers, des primitives de communication qu'il peut utiliser entre processus légers (directement de fil à fil ou via des boîtes aux lettres ou des pointeurs globaux) et de primitives de création de processus légers à distance. Parmi ces environnements on trouve Chant, Nexus, MPI-F et Athapascan-0b.

La deuxième catégorie d'environnements présente un modèle plus homogène de programmation parallèle à l'aide de processus légers. Les fonctionnalités de communication et de multiprogrammation légère sont mieux intégrées et il y a une virtualisation plus poussée de la machine, dans laquelle la notion de tâche (conteneur de processus

légers) tend à disparaître au profit du processus léger comme unique brique de base dans la construction du programme parallèle. Parmi les environnements décrits dans ce chapitre, ceux qui entrent plutôt dans cette deuxième catégorie sont TPVM, PM2, DTMS et Athapascan-0a.

L'avantage d'exposer au programmeur les fonctionnalités de base de la multiprogrammation et des communications est double. Le programmeur habitué à la programmation parallèle "classique" (sans multiprogrammation légère) n'est pas désorienté face à de tels systèmes, et peut migrer ses applications à son rythme vers la multiprogrammation légère, en ne modifiant dans un premier temps que les parties les plus susceptibles de profiter d'une telle technique. Il dispose de plus des fonctionnalités habituelles de communications entre tâches ainsi que des fonctionnalités habituelles de multiprogrammation légère sur une tâche donnée. Le deuxième avantage concerne l'efficacité d'une telle approche. En ne masquant pas les mécanismes de base on n'induit pas de coûts supplémentaires (parfois cachés) à l'exécution. De plus, le coût des primitives est clairement affiché (une synchronisation locale est moins coûteuse qu'un envoi de message par exemple), ce qui permet au programmeur d'adapter sa programmation. Cette démarche de coûts clairement affichés n'est pas nouvelle, elle existe aussi en programmation classique pour aider un programmeur à choisir par exemple entre l'utilisation de la mémoire centrale et des fichiers temporaires [6].

Si tous les environnements décrits offrent sous une forme ou une autre l'appel de procédure ou de service à distance, certains ne permettent pas de communications hors de ce mécanisme (PM2, Athapascan-0a). Ce choix est motivé par la volonté d'offrir la "bonne" méthode au programmeur pour exprimer son algorithme. Une alternative est possible, celle consistant à proposer au programmeur un ensemble d'outils et lui laisser le choix. Il existe alors plusieurs possibilités d'exprimer un même algorithme, mais n'est-ce pas une pratique courante en informatique ?

Enfin, on trouve deux options pour la réalisation des environnements : l'utilisation de bibliothèques standard (ou du moins largement diffusées) et le développement sur mesure. Les deux approches ont leurs point forts et points faibles. L'utilisation de bibliothèques existantes permet de réduire les durées de développement et dans les cas de standards, assure une disponibilité du produit sur une grande palette d'architectures. Le développement sur mesure peut contribuer à améliorer la performance de l'environnement et lui permettre de disposer de fonctionnalités autrement impossibles à obtenir. L'inconvénient de cette approche est l'augmentation de la durée de développement, l'apparition de bogues (dans les bibliothèques largement diffusées un grand nombre de bogues a déjà été corrigé), la nécessité d'effectuer un portage vers chaque nouvelle architecture et le risque de ne pas pouvoir exploiter pleinement certaines d'entre elles (cas de bibliothèques de processus légers en espace utilisateur ne pouvant exploiter efficacement les SMP).

Il nous paraît que les fonctionnalités présentes dans les bibliothèques standard de

6. Ce qui n'empêche pas le système de "tricher" un peu par moments, en cachant des fichiers en mémoire ou en paginant de la mémoire sur le disque.

communication et de processus légers suffisent à implanter un noyau associant communications et multiprogrammation légère et disposant des fonctionnalités nécessaires à une expression aisée de programmes parallèles. C'est le choix que nous avons fait pour Athapascan-0b.

Chapitre 5

Présentation d'Athapascan-0b

5.1 Objectif

Athapascan-0b [8] est un noyau exécutif (*runtime*) permettant l'exécution de programmes parallèles irréguliers à grain fin, comportant souvent un nombre d'unités d'exécution [1] supérieur au nombre de processeurs disponibles dans la machine. Pour assurer une exécution efficace de tels programmes, le mécanisme de découpe doit être peu coûteux, Athapascan-0b fait donc appel aux processus légers. Un algorithme parallèle irrégulier s'exprime alors comme un **réseau dynamique de processus légers communicants** s'exécutant sur une machine parallèle. L'objectif d'Athapascan-0b est triple :

1. Fournir des opérateurs efficaces permettant d'exploiter le parallélisme de la machine site de l'exécution. C'est l'objectif d'**efficacité**.
2. Permettre une expression aisée de la décomposition parallèle à l'aide d'opérateurs adaptés aux schémas classiques de décomposition parallèle. C'est l'objectif de **fonctionnalité**.
3. Être facilement **portable**.

Avant de développer ces trois points (voir chapitres 6 et 7), il convient de définir ce qu'est une machine parallèle ainsi que les types de parallélisme que l'on peut y trouver. Plutôt que de mettre en avant les différentes architectures des machines parallèles (réseaux de stations, machines dédiées, SIMD : *Single Instruction Multiple Data*, MIMD : *Multiple Instruction Multiple Data*, SMP : *Symmetric Multi-Processor* etc...), caractérisons celles-ci suivant les formes de parallélisme que l'on peut y trouver. Quatre types de parallélisme exploitables peuvent être présents dans une machine parallèle :

– Parallélisme inter-nœuds,

1. Le terme "tâche" aurait convenu à merveille ici, mais il a un autre sens tout au long du document...

– Parallélisme intra-nœud entre différents processeurs de calcul,

– Parallélisme sur un nœud entre calculs et communications,

– Parallélisme des communications : sur un nœud, quand celui-ci comporte plusieurs processeurs de communication et sur le réseau, où plusieurs communications peuvent progresser en parallèle en empruntant des routes disjointes.

Athapascan-0b tient compte de ces différentes formes de parallélisme pour atteindre ses objectifs. Les fonctionnalités d'Athapascan-0b (voir section 5.4 et [7, 6]) lui permettent d'exploiter les formes de parallélisme énumérées ci dessus, quand celles-ci sont présentes sur la machine cible.

Dans la suite de cette section, on suppose que sur chaque nœud ("nœud" désignant un monoprocesseur ou l'ensemble des processeurs d'un SMP) ne s'exécute qu'une seule tâche. Cette hypothèse simplifie le discours en permettant d'employer "nœud" à la place de "tâche" pour mettre en avant les parallélismes dans l'exécution. À partir du début de la section 5.2, "tâche" et "nœud" reprennent leurs significations habituelles telles qu'employées tout au long de ce document. La contrainte d'une seule tâche par nœud n'est normalement pas nécessaire à Athapascan-0b. En pratique, quand le réseau et la bibliothèque de communication le permettent, il est tout à fait possible de placer plusieurs tâches sur un même nœud. Un tel fonctionnement peut être intéressant pendant le développement d'une application, si l'on ne dispose pas d'une machine parallèle.

Parallélisme inter-nœuds

Chaque nœud exécute une copie du noyau exécutif Athapascan-0b et une copie du programme applicatif. Les nœuds progressent dans leurs exécutions indépendamment les uns des autres (aux synchronisations dues aux communications près). Le parallélisme inter-nœud est donc trivialement exploité.

Parallélisme intra-nœud

Le parallélisme intra-nœud existe dans une machine quand un nœud comporte plus d'un processeur de calcul ou quand un processeur de calcul permet d'exécuter plusieurs fils d'exécution simultanément. Athapascan-0b exécutant de multiples fils d'exécution sur chaque nœud exploitera ce parallélisme quand il est présent, car les fils s'exécutant sur le nœud se répartiront les processeurs disponibles.

Parallélisme entre calculs et communications

Quand un processeur de communication est présent dans une machine parallèle (ce qui est le plus souvent le cas aujourd'hui), le processeur de calcul initie la communication qui est ensuite prise en charge par le processeur de communication, le processeur

de calcul pouvant alors continuer son exécution. Athapascan-0b permet d'exploiter ce mécanisme à l'aide d'une part les communications asynchrones et d'autre part la multiprogrammation légère. Les communications asynchrones permettent à un fil d'initier une communication et de continuer son exécution avant que la communication ne termine. La présence de multiples fils d'exécution sur un même processeur permet à l'un d'entre eux d'attendre une communication pendant qu'un autre calcule.

Parallélisme des communications

Quand un nœud dispose de plusieurs processeurs de communication, Athapascan-0b permet de les exploiter à l'aide de communications asynchrones ou de plusieurs fils d'exécution. Différentes communications asynchrones émises par un même fil peuvent être traitées par différents processeurs de communication, comme peuvent l'être celles émises par différents fils d'exécution.

Quand le réseau permet à plusieurs communications de progresser en parallèle, Athapascan-0b exploite cette propriété. En effet, un fil peut émettre plusieurs communications "simultanées" (asynchrones) et différents fils d'exécution d'un même nœud peuvent en faire autant. Les communications issues de différents nœuds peuvent également progresser en parallèle sur le réseau.

5.2 Structure et concepts

Le paragraphe suivant décrit de façon très succinte des concepts d'Athapascan-0b qui sont ensuite développés dans des sections séparées.

Une machine parallèle Athapascan-0b est un ensemble de **tâches** exécutant chacune le noyau exécutif Athapascan-0b. Dans chacune de ces tâches s'exécute le **fil principal** ainsi que d'autres fils, créés localement pour exécuter une fonction particulière du programme – on les nomme alors "esclaves" –, ou à distance pour exécuter un **service**. Une autre forme d'exécution à distance est le **service urgent**, dans laquelle l'exécution à distance n'entraine pas la création d'un nouveau processus léger mais est prise en charge par un fil démon (*daemon*) d'Athapascan-0b. Les services et les services urgents doivent être déclarés lors de la phase d'**initialisation** de la machine parallèle par le fil principal. Les fils présents dans une même tâche peuvent se **synchroniser** entre eux par les primitives de synchronisation habituelles. Des fils sur deux tâches quelconques peuvent communiquer par envoi de messages, soit en décrivant les données à l'aide de "types de données" (*datatypes*) soit en emballant les données dans des **tampons**. La destination d'un message sur une tâche distante s'appelle le **port**. Les opérations d'envoi et de réception de messages, ainsi que toutes les opérations potentiellement longues d'Athapascan-0b, possèdent une version **bloquante** et une version non-bloquante ou **asynchrone**. La figure 5.1 illustre certains des concepts évoqués ci-dessus et développés ci-après.

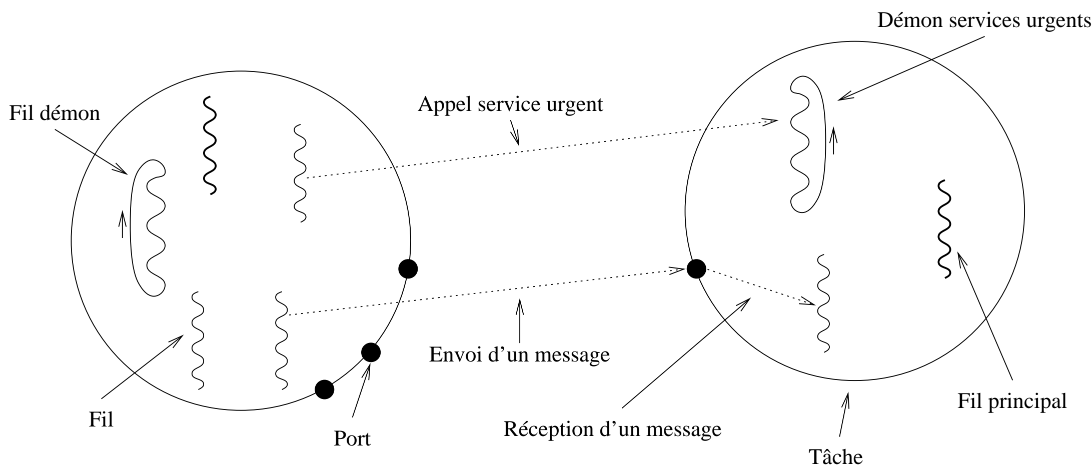

FIG. 5.1 – *Deux tâches dans une machine parallèle Athapascan-0b.*

5.2.1 Tâches

La machine parallèle Athapascan-0b est composée de tâches, numérotées consécutivement à partir de zéro. Chaque tâche est un processus lourd exécutant l'application liée (*linked*) à l'exécutif Athapascan-0b, la machine fonctionnant en mode SPMD (*Single Program Multiple Data*). Ce processus lourd abrite des processus légers. Parmi ces processus légers s'en trouve un particulier, le fil initial, qui a été créé au démarrage du processus lourd, et qui exécute la fonction `main` du programme.

Les différentes tâches composant une machine parallèle Athapascan-0b sont placées sur des nœuds (machines) physiques. Selon l'architecture de la machine cible et la bibliothèque de communication utilisée, il est possible ou pas de placer plusieurs tâches Athapascan-0b sur le même nœud physique. Le fait que des tâches soient placées sur un même nœud est absolument transparent au programmeur. Les nœuds ne sont pas nécessairement identiques : certains peuvent être des monoprocesseurs et d'autres des SMP : Athapascan-0b supporte leur hétérogénéité.

5.2.2 Services

Chaque tâche, en plus de son point d'entrée initial (`main`), comporte des fonctions pouvant être appelées depuis les autres tâches de la machine : les services. Afin que les services aient des noms cohérents sur toutes les tâches, ils sont tous déclarés durant la phase d'initialisation de la machine par le fil principal de chaque tâche.

Tout fil de toute tâche peut appeler un service sur une autre tâche. Il spécifie le numéro de la tâche concernée et le nom du service à appeler, et peut alors créer sur la tâche distante un nouveau processus léger, exécutant la fonction du service. L'appel de service est le mécanisme d'Athapascan-0b permettant de créer à distance des fils d'exécution.

Un fil peut également choisir d'effectuer un appel de service urgent sur une autre

tâche. Il spécifie comme précédemment le nom du service, mais sur la tâche distante il n'y aura pas création d'un nouveau fil d'exécution. La fonction du service sera exécutée par un fil dédié de l'exécutif d'Athapascan-0b. Ce mode d'appel à distance permet d'exécuter une fonction sur une tâche distante avec un surcoût plus faible qu'un appel de service classique. Du fait de l'exécution du service par un fil de l'exécutif, certaines limitations sont imposées à un service urgent afin de prévenir les blocages.

5.2.3 Démarrage et initialisation

La machine parallèle est prête à commencer son exécution quand sur les nœuds ont été créées les tâches Athapascan-0b, c'est à dire les processus lourds intégrant le programme applicatif et l'exécutif Athapascan-0b. Cette étape d'initialisation de l'exécution sur les nœuds ne fait pas partie d'Athapascan-0b et elle est gérée par l'environnement d'exécution de la machine, le plus souvent associé à la bibliothèque de communication sur laquelle est construit Athapascan-0b.

Au démarrage de la machine parallèle, chaque tâche exécute le fil d'exécution principal (*main thread*, celui qui exécute la fonction `main` du programme), qui est responsable de l'initialisation de la machine (déclaration des services pour que ceux-ci puissent être appelés depuis d'autres nœuds et d'autres initialisations).

5.2.4 Fils et synchronisation

Chaque tâche est un processus lourd abritant des processus légers. Initialement, chaque tâche ne comporte que le fil principal, qui "naît" automatiquement à la création de la tâche. Les fils principaux des différentes tâches créent ensuite d'autres fils localement ou à distance. Dans une création locale (dans la même tâche), le nouveau fil créé se nomme "esclave" dans la terminologie Athapascan et peut exécuter n'importe quelle fonction du programme. Une création à distance se faisant par appel de service, seules les fonctions des services déclarés durant l'initialisation peuvent être appelées. Les fils en exécution sur une tâche ont accès aux mêmes fonctionnalités qu'ils aient été créés localement ou à distance : ils peuvent utiliser les primitives habituelles de synchronisation (*mutex*, sémaphores etc...).

5.2.5 Ports

Les fils d'exécution n'étant "visibles" qu'à l'intérieur d'une tâche et pouvant avoir des durées de vie extrêmement brèves (en rapport avec le grain – fin – de l'application), il n'est ni aisé ni souhaitable de définir un mécanisme permettant à un fil d'une tâche d'envoyer un message à un fil particulier d'une autre tâche (voir section 4.1). Pour permettre une flexibilité dans la gestion des messages en n'interdisant bien sur pas les communications entre deux fils particuliers, la notion de port a été introduite.

Un port est un réceptacle pour les messages arrivant sur une tâche. Tout message envoyé à une tâche spécifie le port de destination dans lequel il attend qu'un fil vienne

le lire (en effectuant une réception sur ce port). Tout fil d'une tâche peut recevoir des messages de tout port de cette tâche. Plusieurs fils peuvent recevoir les messages arrivant à un même port. Un message reçu sur ce port sera alors reçu par un et un seul de ces fils (l'ordre des réceptions correspond à l'ordre d'arrivée des messages). Les ports doivent être déclarés lors de la phase d'initialisation ou plus tard dans l'exécution du programme afin que les tâches puissent s'entendre sur un système de nommage cohérent.

À chaque message adressé à un port est associée une étiquette. Un fil effectuant une réception sur un port précise l'étiquette des messages qu'il désire recevoir. Ceci permet à l'application de multiplexer sur un port plusieurs flux de messages.

5.2.6 Tampons

L'envoi et la réception de données peut se faire de deux façons différentes. Les données peuvent être envoyées directement à partir de leur emplacement en mémoire, sans recopie, quand leur structure en mémoire est "simple". Dans le cas contraire, les données sont d'abord "emballées" (*packed*) dans un tampon (*Buffer*) puis celui-ci (zone contiguë en mémoire) est envoyé dans un message. Lorsqu'un envoi est fait à l'aide d'un tampon, la réception correspondante doit l'être aussi.

Les tampons sont également utilisés quand plusieurs données doivent être agrégées pour être émises dans un seul message. C'est le cas notamment lors des appels de service ou de service urgent, où le tampon contient le nom du service à appeler ainsi que des paramètres passés lors de l'appel.

5.2.7 Types de données

Les "types de données" (*datatypes*) permettent de décrire le type et l'organisation en mémoire de données. Des *datatypes* existent pour les types de base (entier, flottant, caractère, octet etc…) et des primitives permettent de construire des organisations de type structures ou tableaux. Ces primitives ne font pas partie stricto sensu d'Athapascan-0b mais d'une extension nommée Athapascan-0b Formats. Les organisations régulières de données en mémoire sont facilement décrites à l'aide de *datatypes*. Une telle description permet aux différentes fonctions de communication d'envoyer les données directement à partir de leur emplacement en mémoire, évitant ainsi des copies coûteuses.

5.2.8 Opérations bloquantes et asynchrones

Les primitives d'Athapascan-0b effectuant des opérations longues ou potentiellement bloquantes existent en deux versions. Une version bloquante dans laquelle le fil effectuant l'opération attend la fin de celle-ci et une version asynchrone, dans laquelle le fil initie l'opération et peut continuer son exécution. Dans ce dernier cas il peut se

bloquer ultérieurement en attente de la fin de l'opération ou simplement tester cette terminaison.

5.2.9 Ordonnancement

Les politiques d'ordonnancement (*scheduling*) disponibles dans le noyau de processus légers utilisé sont remontés à l'utilisateur d'Athapascan-0b. Quand le noyau dispose de priorités, Athapascan-0b se réserve les priorités extrêmes et rend les autres disponibles à l'utilisateur.

Les services disposent d'une priorité par défaut (choisie lors de la déclaration du service), mais peuvent être lancés en spécifiant une priorité particulière. Lors de la création d'un fil esclave, le créateur spécifie également la priorité à donner à l'esclave. Il n'y a pas de gestion globale des priorités, et chaque priorité ou type d'ordonnancement est relatif aux autres processus légers de la même tâche.

5.3 interface de programmation

L'interface de programmation d'Athapascan-0b se présente sous la forme d'une bibliothèque de fonctions,[2] à utiliser depuis un langage de programmation de type C ou C++. Contrairement à d'autres systèmes (Athapascan-0a par exemple), aucune nouvelle structuration du code source n'est imposée.

La philosophie d'Athapascan-0b cherche une exploitation optimale de toutes les formes de parallélisme présentes dans une machine et de ce fait les primitives d'Athapascan-0b (les fonctions de la bibliothèque) n'engendrent qu'un blocage minimal des processus légers qui les invoquent. Plus précisément, quand une opération est potentiellement longue, car impliquant des communications, il est possible de l'effectuer en deux temps. Dans un premier temps l'opération est initiée et le processus léger continue son exécution sans blocage. Dans un deuxième temps le processus léger peut vérifier la terminaison de l'opération, sous forme d'un simple test non bloquant ou d'une attente. Athapascan-0b se conforme systématiquement à ce modèle dissociant l'initiation d'une action de sa terminaison. Une "requête" (type `a0tRequest`) permet de faire la correspondance entre une action d'initiation et l'attente correspondante :

```
/* Initiation d'une opération */
a0InitSomething(..., &request);

...le fil continue son exécution...

/* Test de la fin d'une opération... */
```

2. Les fonctions d'Athapascan-0b renvoient toutes un code indiquant la réussite de l'appel ou un code d'erreur. Par souci de lisibilité, les tests de ce code de retour ne sont pas présentés dans les exemples de ce document.

```
a0TestRequest(&request, &result);
/* ...ou attente de la fin d'une opération */
a0WaitRequest(&request);
```

Cette approche, en plus du fait qu'elle permet aux fils de ne pas être bloqués inutilement et de pouvoir ainsi exécuter en parallèle des opérations non conflictuelles (recouvrement de communications par des calculs par exemple) a un autre avantage en ce qu'elle permet à Athapascan-0b d'enchaîner l'exécution de différentes opérations dans l'ordre de disponibilité des ressources physiques :

```
/* Initiation de trois requêtes */
a0InitSomething(..., &request1);
a0InitSomething(..., &request2);
a0InitSomething(..., &request3);

...le fil continue son exécution...

/* Attente des trois requêtes */
a0WaitRequest(&request1);
a0WaitRequest(&request2);
a0WaitRequest(&request3);
```

Dans ce code, si par exemple les ressources nécessaires à l'opération correspondante à `request3` sont libres alors que celles nécessaires aux autres opérations ne le sont pas, la troisième opération sera exécutée avant les deux autres. L'exécution des trois opérations de cette façon peut donc être plus rapide que leur exécution dans l'ordre imposé par le programmeur (`request1`, `2` puis `3`). Les fonctions pouvant bénéficier d'une telle approche disposent de deux versions, l'une non bloquante (asynchrone ou immédiate), dans laquelle l'initiation de l'action est séparée de sa terminaison et l'autre bloquante, la fonction ne retournant qu'une fois l'opération terminée. Sauf mention contraire, les exemples de code dans la suite ne présentent que les versions bloquantes.

La plupart des identificateurs d'Athapascan-0b suivent une convention de nommage. Leur nom est composé d'un préfixe permettant de connaître la nature de l'identificateur (fonction, type ou constante), suivi d'un nom précisant son rôle. Le préfixe est `a0` pour les fonctions, `a0t` pour les types et `A0` pour les constantes. La suite de l'identificateur est une concatenation de mots ou de caractères donnant le rôle de l'identificateur. La première lettre de chaque mot est majuscule et les autres minuscules. Sont exclus de cette convention les identificateurs des *datatypes*, qui s'inspirent de la convention de nommage de MPI.

Les objets Athapascan-0b sont alloués par le programmeur et sont passés aux fonctions de la bibliothèque par référence. Cette approche permet de simplifier la bibliothèque, car elle n'a pas à gérer l'allocation (sauf pour les tampons, voir section 5.4.2),

et laisse au programmeur le soin d'allouer les objets selon la méthode la mieux adaptée à son application. Une allocation dynamique des objets dans la pile du processus léger n'est envisageable que si l'on peut assurer que l'objet disparaîtra avant la fin de la fonction l'ayant alloué.

Les programmes Athapascan-0b doivent inclure le fichier `a0b.h` contenant les définitions des constantes, des types et des fonctions de la bibliothèque.

5.4 Fonctionnalités

Cette section présente les fonctionnalités d'Athapascan-0b, et donne des exemples d'utilisation.

5.4.1 Initialisation et terminaison

Le lancement des exécutables constituant les codes Athapascan-0b sur la machine parallèle ne fait pas partie d'Athapascan-0b et est pris en charge par l'environnement de programmation de la bibliothèque MPI utilisée.

Quand les tâches Athapascan-0b démarrent sur tous les nœuds, elles doivent effectuer une série d'initialisations avant de commencer à exécuter le code de l'application proprement dit. Ces initialisations concernent surtout la déclaration des services exportés mais peuvent aussi concerner d'autres initialisations, comme par exemple la création des *ports* et l'initialisation des bibliothèques utilisées. Toutes les tâches exportent les services dans le même ordre (elles exécutent le même code d'initialisation), ce qui permet d'attribuer aux services des numéros cohérents sur l'ensemble de la machine parallèle. L'initialisation d'Athapascan-0b s'effectue en plusieurs temps. Dans un premier temps un début d'initialisation (`a0Init`) met en place les structures destinées à accueillir les déclarations des services et récupère les arguments de ligne de commande passés au lancement entre autres actions. La tâche exporte ensuite les services pouvant être appelés à distance et effectue d'autres initialisations. La fin de l'initialisation (`a0InitCommit`) termine les opérations d'initialisation et démarre les démons nécessaires à Athapascan-0b. En fin d'exécution, un autre appel (`a0Terminate`) sert à terminer l'exécution d'Athapascan-0b en arrêtant proprement les démons puis en terminant MPI. La terminaison de l'application parallèle a lieu quand **toutes** les tâches ont effectué l'appel à la fonction de terminaison. L'initialisation et la terminaison se font dans le fil principal des tâches. L'exemple suivant donne le squelette d'un programme exportant un service dont la fonction se nomme `fctServ`, l'identificateur associé au service et qui servira pour les appels est `idServ`.

```
#include <a0b.h>

/* Définition d'un service */
int idServ;
```

```
a0tError fctServ(a0tBuffer *data) {
  ...corps du service...
}

/* Fil d'exécution principal */
int main(int argc, char **argv) {
  a0Init(&argc, &argv);
  a0NewService(&idServ, fctServ, ...);
  ...d'autres initialisations éventuellement :
     ports, bibliothèques etc...
  a0InitCommit();

  ...corps du programme principal...

  a0Terminate();
}
```

Quand des bibliothèques parallèles sont utilisées dans un programme Athapascan-0b, elles doivent initialiser leurs services, leur ports de communications etc... tout comme le programme principal. Dans ce cas, chaque bibliothèque fournit une fonction d'initialisation qui est appelée par le programme principal en même temps que ses propres initialisations. Voici un exemple de fonction d'initialisation de bibliothèque :

```
extern void ma_lib_init() {
  /* Déclaration des services de la bibliothèque */
  a0NewService(&libIdServ1, libFctServ1, ...);
  a0NewService(&libIdServ2, libFctServ2, ...);
  /* Création d'un port de la bibliothèque */
  a0NewPort(&libPort, ...);
  ...autres initialisations...
}
```

5.4.2 Création et utilisation de tampons

Les tampons d'Athapascan-0b (*buffers*) sont un des moyens d'envoyer des données dans des messages et le seul moyen d'envoyer des paramètres à un appel de service à distance (urgent ou normal). L'allocation de l'objet tampon (type `a0tBuffer`) est à la charge du programmeur mais l'allocation de l'espace de stockage du tampon se fait par un appel à la bibliothèque (`a0NewBuffer`) en précisant la taille maximale (en octets) du tampon. Cet appel permet aussi de préciser le type du tampon (envoi et réception de message, appel de service ou encore appel de service urgent). Une fois le tampon créé et son espace de stockage alloué, le programme emballe les données nécessaires dans le tampon avant d'envoyer celui-ci dans un message. Coté récepteur,

une fois un tampon reçu, le programme déballe les données pour pouvoir les utiliser. L'emballage et le déballage des données entraînent des copies depuis le tampon vers la zone mémoire des données ou dans l'autre sens. Voici un exemple d'utilisation de tampon pour l'envoi d'un message :

```
a0tBuffer tampon;
float tabFlottant[10];
int entier;

/* Allocation de l'espace de stockage */
a0NewBuffer(&tampon, A0SendBufferType, TAILLEMAX);

/* Emballage des données */
a0Pack(&tampon, a0_FLOAT, tabFlottant, 10);
a0Pack(&tampon, a0_INT, &entier, 1);

/* Envoi du tampon */
a0SendBuffer(&port, destination, tag, &tampon);
```

5.4.3 Déclaration et appel de service

Les services, urgents et normaux, doivent être déclarés par toutes les tâches dans le même ordre, afin d'être identifiés de façon cohérente. À la déclaration d'un service une valeur entière est retournée dans un identificateur qui sert par la suite à faire les appels distants à ce service. Le service admet comme seul paramètre un tampon contenant les arguments envoyés par l'appelant. La déclaration d'un service admettant en paramètre deux flottants et un entier ressemble à ceci :

```
a0tError monService(a0tBuffer *data) {
  float flottants[2];
  int entier;

  /* déballage des arguments */
  a0Unpack(data, a0_Float, flottants, 2);
  a0Unpack(data, a0_INT, &entier, 1);

  ...corps du service...
}
```

Pour pouvoir être appelé, ce service doit être déclaré durant la phase d'initialisation :

```
a0NewService(&idService, monService, sched, prio, stack);
```

`idService` est un entier, `sched` est le mode d'ordonnancement par défaut à utiliser pour le service, `prio` est sa priorité par défaut et `stack` la taille par défaut de sa pile. L'appel de ce service se passe alors comme suit : (on suppose les variables déjà déclarées et le tampon déjà alloué du type correspondant à l'appel)

```
/* Emballage des paramètres */
a0Pack(&tamponAppel, a0_FLOAT, &f1, 1);
a0Pack(&tamponAppel, a0_FLOAT, &f2, 1);
a0Pack(&tamponAppel, a0_int, &i, 1);

/* Appel de service normal... */
a0StartRemoteThread(noeud, idService, sch, pri, st,
                    &tamponAppel);
/* ...ou appel de service urgent */
a0StartRemoteUrgent(noeud, idService, &tamponAppel);
```

Dans l'appel de service normal, les paramètres `sch`, `pri` et `st` indiquent respectivement le type d'ordonnancement, la priorité et la taille de la pile du fil créé (il est possible d'indiquer que les valeurs par défaut données lors de la déclaration du service doivent être utilisées). L'appel par `a0StartRemoteUrgent` est réservé aux services non susceptibles de se bloquer.

5.4.4 Création locale de processus légers

Si les services permettent de créer des processus légers à distance, la création d'un processus léger local, appelé "esclave" en Athapascan-0b implique moins de surcoût car il n'y a ni passage de message ni utilisation d'un tampon, et permet une plus grande souplesse d'utilisation : il est possible de se synchroniser avec la terminaison du fil. Profil d'une fonction pouvant être exécutée par un processus léger :

```
a0tError maFonction(void *arg) {
  ...corps de la fonction...

  /* Renvoi d'un valeur de retour */
  a0ExitThread(res);
}
```

Les fonctions exécutées dans un fil d'exécution séparé admettent un argument scalaire et peuvent renvoyer un résultat scalaire (le type `void *` est utilisé pour permettre le passage de n'importe quel type scalaire). La création d'un fil exécutant cette fonction et l'attente de la terminaison de ce fil se font de la façon suivante :

```
a0tThread fil;

a0NewSlave(&fil, sch, pri, st, maFonction, &parametre);
...
a0JoinSlave(&fil, &resultat);
```

`parametre` et `resultat` sont de type scalaire et comme précédemment, `sch`, `pri` et `st` indiquent respectivement le type d'ordonnancement, la priorité et la taille de la pile du fil créé (des valeurs par défaut sont également disponibles).

5.4.5 Synchronisation locale

Des fils s'exécutant dans la même tâche peuvent se synchroniser à l'aide de *mutex* ou de sémaphores. Les fonctionnalités habituelles sont implantées :

```
a0tMutex monMutex;
a0tSemaphore monSemaphore;

/* Initialisation */
a0NewMutex(&monMutex);
a0NewSemaphore(&monSemaphore, valeurInitiale);

/* Verrouillage ou acquisition */
a0LockMutex(&monMutex);
a0PSemphore(&monSemaphore);

/* Déverrouillage ou libération */
a0UnlockMutex(&monMutex);
a0VSemaphore(&monSemaphore);

/* Essai d'acquisition non bloquant */
a0TryPSemaphore(&monSemaphore, &acquis);
```

`acquis` est un booléen indiquant si le sémaphore a pu être acquis ou pas.

5.4.6 Création de port

Il y a deux modes de création de ports. Dans le premier, l'appel de création doit être fait sur tous les nœuds de la machine parallèle et tous les appels renvoient un port identique. Ce type d'appel est utile pour créer des ports "bien connus"[3] pouvant être utilisés par toutes les tâches sans nécessiter de diffusion préalable. La contrainte de ce type de création est que toutes les tâches de la machine doivent créer les ports dans le

3. À rapprocher du concept de *well-known services* d'Internet.

même ordre (comme pour les déclarations de services). On parle alors de "port global". Lors de la création d'un port il est nécessaire de spécifier le nombre d'étiquettes différentes allant être utilisées avec le port. Voici un exemple de création d'un port global pouvant être utilisé avec cinq étiquettes différentes :

```
a0tPort portGlobal;

a0NewPort(&portGlobal, 5, A0GlobalCreation);
```

L'autre type de port est le port local, qui n'est créé que par une seule tâche effectuant l'appel de création. L'unicité du port créé est garantie sur l'ensemble de la machine parallèle, pour l'exécution entière. Afin d'être utilisé dans des communications, un port local doit être communiqué à un ou d'autres tâches (dans un message) qui peuvent ensuite l'utiliser. La création d'un port local avec une étiquette possible se fait comme suit :

```
a0NewPort(&portLocal, 1, A0LocalCreation);
```

5.4.7 Envoi et réception de messages

Les messages dans Athapascan-0b sont envoyés par un fil d'exécution vers un port d'une tâche donnée avec une étiquette donnée. Sur la tâche destination, tout fil effectuant une réception sur le port en question avec l'étiquette correspondante, sera en mesure de recevoir le message. Si plusieurs fils sur la tâche destination tentent la même réception, le premier d'entre eux recevra le message.

Il existe deux types de primitives d'envoi et de réception de messages. L'un permet d'envoyer directement les données depuis leur emplacement mémoire et l'autre passe par un "emballage" des données dans un tampon. Chaque type de primitive possède deux variantes, bloquante et immédiate (comme toutes les opérations potentiellement bloquantes dans Athapascan-0b).

```
/* Envoi d'un tampon */
a0SendBuffer(&port, dest, tag, &tampon);

/* Envoi direct */
a0Send(&port, dest, tag, typeDonnees, adresse, nb);
```

`port`, `dest` et `tag` déterminent la destination du message et l'étiquette associée. Dans le premier cas, le tampon sera envoyé et devra être reçu par une primitive de réception de tampon. Dans le deuxième cas, `typeDonnees` est le type élémentaire de la donnée, `adresse` est l'adresse de début des données et `nb` est le nombre de données élémentaires à envoyer. Un message envoyé directement doit être reçu par une réception de même type.

```
/* Réception d'un tampon */
a0ReceiveBuffer(&port, source, tag, &req, &tampon);

/* Réception directe */
a0Receive(&port, source, tag, &req,
                          typeDonnees, adresse, nb);
```

Dans les opérations de réception, il est obligatoire de spécifier exactement le port et l'étiquette de la réception (il n'est pas possible de demander une réception "avec toute étiquette"). Par contre `source`, la tâche du fil originaire du message peut prendre la valeur `A0AnySource` pour recevoir un message venant de n'importe quel fil de n'importe quelle tâche. `typeDonnees`, `adresse` et `nb` décrivent la destination des données en mémoire, leur type et leur nombre. Le paramètre `req` permet de connaître exactement le nombre d'éléments reçus (pouvant être inférieur au nombre d'éléments souhaité) ainsi que la tâche originaire du message.

5.5 Démarche générale d'utilisation

Lors du démarrage de la machine parallèle, s'exécute dans chaque tâche le fil d'exécution principal. Ce fil s'occupe des diverses initialisations de la machine : déclaration des service appelables à distance, création des ports de communication, initialisation des bibliothèques utilisées etc… Après l'initialisation, les fils principaux peuvent créer des processus légers "démons" assurant des services pour le compte des autres fils, locaux ou distants. Un processus démon se bloque en réception sur un port de communication pour servir des requêtes à distance ou sur une synchronisation locale pour servir les fils d'exécution de sa tâche. Lors d'une requête distante à un démon, l'appelant peut passer dans le message le port sur lequel le démon répondra à la requête. L'appel à un démon est plus rapide que l'appel de service à distance car n'impliquant pas de création de fil d'exécution et est moins contraignant que l'appel de service urgent. Les fils principaux peuvent également créer les objets de synchronisation nécessaires aux fils s'exécutant dans la tâche.

Suite à ces actions, que l'on peut globalement considérer comme l'initialisation de la machine parallèle et de l'application, deux choix sont possibles. Dans le premier, correspondant à un modèle de programmation où un fil "maître" orchestre le déroulement de l'application, tous les fils principaux sauf un cessent de s'exécuter, le fil restant installant le calcul parallèle sur la machine à l'aide d'appels de service à distance accompagnés éventuellement de création de fils locaux. Dans le deuxième choix, tous les fils ou un certain nombre d'entre eux continuent leur exécution, l'installation des calculs étant alors moins centralisée et plus symétrique.

Lors de l'exécution de l'application, s'exécutent dans chaque tâche des fils ayant été crées localement (des esclaves) ou à distance (des services). Ces fils se synchronisent entre eux par les *mutex* et les sémaphores d'Athapascan-0b. Ils peuvent égale-

ment partager l'utilisation des ports pour la réception des messages. Un tel fonctionnement est courant quand un fil exécutant un service crée localement des esclaves à qui il délègue une partie de son travail et qui peuvent de ce fait être amenés à partager des ports.

L'application se termine quand les fils principaux des toutes les tâches se terminent, en effectuant l'appel de terminaison d'Athapascan-0b. C'est au programmeur d'assurer que cette terminaison correspond bien à la fin de l'application. Il faut notamment veiller à ce que les fils principaux ne terminent pas alors que d'autres fils sont encore en train d'effectuer des traitements utiles, car ces derniers cesseraient purement et simplement d'exister.

Chapitre 6

Réalisation

6.1 Problématique

6.1.1 Objectif

Plusieurs objectifs peuvent être poursuivis lors d'une intégration de communications et de multiprogrammation légère. Certains de ces objectifs concernent la richesse des fonctionnalités offertes, d'autres la performance. Dans le contexte du calcul parallèle, nous avons choisi cette dernière option, à savoir d'offrir un support d'exécution multiprogrammé ne pénalisant ni les communications ni les calculs par rapport à un environnement monoprogrammé. Plus précisément, l'on souhaite que les performances d'un processus léger communiquant soient identiques à celles d'un processus communiquant en environnement monoprogrammé. Il s'agit de présenter la même latence et le même débit dans les communications. On souhaite également ne pas imposer un surcoût trop grand à l'exécution du fait de la gestion des communications en environnement multiprogrammé, afin de garder des performances en calcul similaires à celles d'un processus communiquant monoprogrammé.

Ces objectifs ne sont pas atteignables tels qu'énoncés, puisque par exemple quand plusieurs processus légers d'un même nœud communiquent simultanément, ils se partagent nécessairement le débit de communication du nœud. Chacun d'entre eux ne peut donc disposer du débit maximal dont disposerait un processus monoprogrammé mais l'on peut souhaiter alors que le débit cumulé des communications de ces fils ne soit pas inférieur au débit de communication d'un processus monoprogrammé. Ces objectifs doivent donc plutôt être considérés comme une ligne directrice, tant pour la réalisation que pour l'évaluation de l'environnement. Différentes contraintes font que l'on en est selon les cas plus ou moins loin.

6.1.2 États d'un fil d'exécution communiquant

La figure 3.2 page 32, présente les transitions d'état d'un fil d'exécution, et par là même les états dans lesquels peut se trouver le fil. Cette figure est valable pour tous

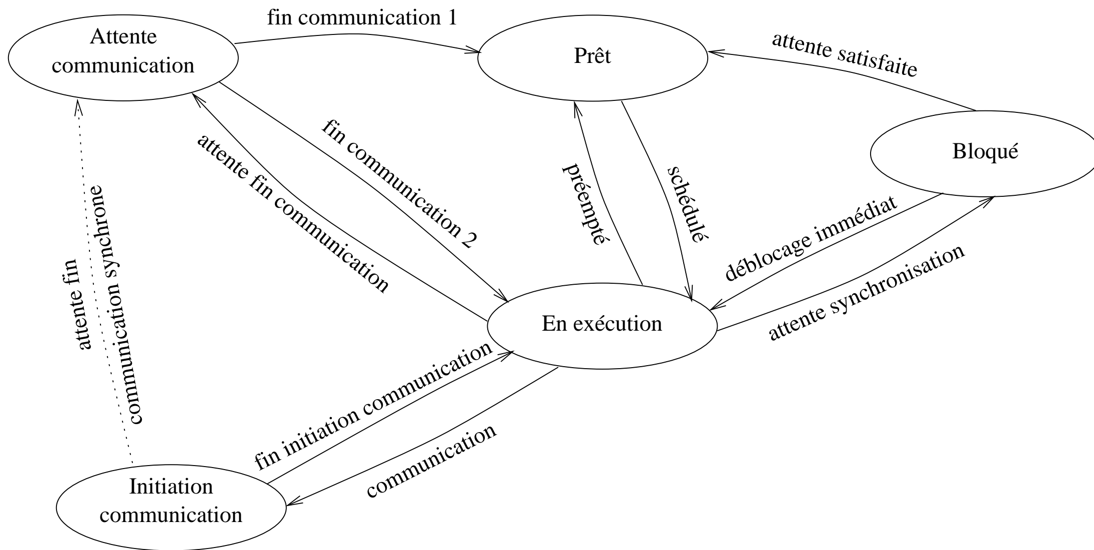

FIG. 6.1 – *Transitions d'état d'un fil d'exécution effectuant des communications.*

les fils d'exécution, et en particulier pour les fils d'exécution effectuant des communications. Mais, pour mieux comprendre le comportement d'un fil d'exécution communiquant, il peut être intéressant de présenter un diagramme plus détaillé, dans lequel certains états sont subdivisés. La figure 6.1 présente un tel diagramme (l'état "Terminé" n'y apparaît pas). Ce diagramme permettra de désigner de façon précise les différentes étapes impliquées dans une opération de communication effectuée par un fil.

L'état "en exécution" de la figure 3.2 recouvre les états "en exécution" et "initiation communication" de la figure 6.1. De même pour l'état "bloqué" de la figure 3.2 qui recouvre les états "bloqué" et "attente communication" de la figure 6.1.

Un fil d'exécution initiant une communication passe de l'état "en exécution" à l'état "initiation communication" puis retourne à l'état "en exécution". Si cette initiation de communication est faite pour le compte d'une communication synchrone (bloquante), le fil passe ensuite immédiatement dans l'état "attente communication", d'où la transition en pointillés entre les états "initiation communication" et "attente communication". Dans le cas d'une communication asynchrone, le fil continue son exécution, puis passera ultérieurement dans l'état "attente communication" pour attendre la fin de la communication. Que ce soit dès l'initiation ou ultérieurement, une fois le fil dans l'état "attente communication", deux choix sont possibles :

– Si la communication termine immédiatement (ou est déjà terminée quand le fil passe dans l'état "attente communication"), le fil peut continuer son exécution (repasser dans l'état "en exécution") sans perdre le processeur. Il est également possible qu'un système fasse une attente active d'une certaine durée (le fil continue à s'exécuter en testant sans arrêt la terminaison de sa communication) pour économiser une commutation de fil dans le cas où la communication se termine

rapidement. Ces cas de figure correspondent à la transition “fin communication 2”.

– La communication n’est pas terminée, le fil se bloque en attente de sa terminaison en restant dans l’état “attente communication”. Il en sortira vers l’état “prêt” quand la communication terminera, et sera ensuite ordonnancé comme les autres fils de l’état “prêt”. Ce cas de figure correspond à la transition “fin communication 1”.

La durée d’une communication synchrone telle que perçue par un fil est le temps qui s’écoule entre le moment où le fil initie la communication (passage dans l’état “initiation communication”) et le moment où le fil peut à nouveau s’exécuter, c’est-à-dire retour à l’état “en exécution” après passage dans l’état “attente communication”.

6.1.3 Intégration efficace

Une intégration efficace de communications et de multiprogrammation légère a entre autres objectifs de présenter à l’application, quand les conditions le permettent, un débit et une latence de communication aussi proches que ceux d’un processus monoprogrammé. Pour atteindre cet objectif, il convient de réduire les délais des différentes transitions d’état d’un fil effectuant une communication.

On ne s’intéresse qu’aux transitions d’état relatives à l’envoi et à la réception de messages. Les autres primitives de communication peuvent être (et souvent sont) construites au dessus de ces deux primitives de base et si ce n’est le cas, le comportement du système et les contraintes associées sont similaires à celles décrites ici.

On considère dans la suite que les communications sont asynchrones : le processus léger initie la communication et continue son exécution. Ultérieurement il peut se bloquer en attente de sa terminaison ou simplement tester son état d’avancement par un appel non bloquant. Une communication bloquante est l’opération d’initiation suivie immédiatement de l’opération d’attente. Cette hypothèse d’appels asynchrones ne fait pas perdre sa généralité au discours. Les deux sections suivantes traitent de l’efficacité des opérations d’envoi et de réception.

Envoi d’un message

Décrivons l’enchaînement des événements dans un système multiprogrammé idéal. Quand un fil initie l’envoi d’un message, celui-ci commence immédiatement à quitter le nœud, à un rythme dicté par la latence, le débit et la charge du réseau. Le fil continue son exécution et peut effectuer d’autres calculs pendant ce temps. Le fil se bloque ensuite sur l’attente de fin d’émission et est débloqué **dès que l’envoi est terminé** (et ce, quel que soit le sens donné ici à “terminé”). En d’autres mots, la transition entre l’état “attente communication” et l’état “prêt” ou “en exécution” se fait au plus tôt, dès que l’envoi est terminé. Le fil peut alors reprendre son exécution immédiatement

ou attendre dans l'état "prêt" qu'un processeur se libère (cette attente n'est pas signe d'inefficacité des communications mais est due au fait que le nombre de processeurs physiques est inférieur au nombre de processus légers prêts à s'exécuter).

Dans un système "réel" (par opposition à "idéal"), l'exploitation du réseau à son débit maximal ainsi que le déblocage d'un fil en attente dès la terminaison d'un envoi peuvent présenter des performances inférieures à l'optimum. Le processeur n'est en effet pas entièrement dédié à cette tâche puisqu'il gère en parallèle l'exécution d'autres processus légers. Voir section 6.4 pour un approfondissement de cette question dans le cas particulier de la réalisation d'Athapascan-0b.

Réception d'un message

Pour la réception d'un message, les contraintes sont similaires à celles de l'envoi. Quand un fil initie la réception, le système doit être prêt à recevoir un message et le stocker dans la zone mémoire indiquée. Lorsqu'un fil est bloqué en attente de terminaison d'une réception, il doit être débloqué dès que la réception se termine, à savoir dès que les données à recevoir sont arrivées sur le nœud et ont été stockées dans les emplacements indiqués lors de l'opération d'initiation de la réception.

On retrouve dans la réception un aspect similaire à l'envoi de message : le système fonctionnant en multiprogrammation légère doit assurer une latence et un débit, en tenant compte de la charge du réseau, similaires à ceux du fonctionnement monoprogrammé.

Un autre aspect de la réception que l'on ne retrouve pas dans l'envoi, est la dépendance entre celle-ci et des événements extérieurs. La transition d'état du fil effectuant la réception entre l'état "attente communication" et l'état "prêt" ou "en exécution" dépend de l'arrivée d'un message. Une arrivée de message est immédiatement portée à la connaissance d'un système "idéal" qui peut immédiatement débloquer le fil en attente de ce message.

Dans un système "réel", l'arrivée d'un message peut être signalée par le réseau par une interruption, afin de faire avancer la communication associée. Un tel mécanisme peut s'avérer coûteux et de plus il existe des plates-formes sur lesquelles il n'existe pas. C'est alors à l'environnement de programmation d'interroger périodiquement le réseau (**scruter** le réseau) afin de découvrir si de nouveaux messages sont arrivés. Dans le deux cas, la durée de l'opération ou sa fréquence font qu'un message n'est pas pris en compte par le système immédiatement à son arrivée.

Exemple de communications

Cette section illustre par l'exemple ce qui se passe lors d'une communication d'un processus monoprogrammé, d'une communication d'un processus léger et de plusieurs communications simultanées de plusieurs processus légers. Les communications décrites sont synchrones (bloquantes). Chaque communication comprend trois phases. Dans la première, le processus effectue les opérations d'initiation de la communica-

tion puis se bloque. Ensuite, dans la deuxième phase, la communication se fait sur le réseau (cette phase n'occupe pas le processeur de calcul). Enfin, quand la communication sur le réseau est finie, le processus peut être débloqué. On parle de "point de déblocage" pour caractériser le moment dans le temps à partir duquel le processus peut prendre connaissance de la fin de sa communication. Dans le cas de processus légers, le déblocage effectif du processus peut être postérieur à ce point, dans la mesure où d'autres processus légers occupent le processeur.

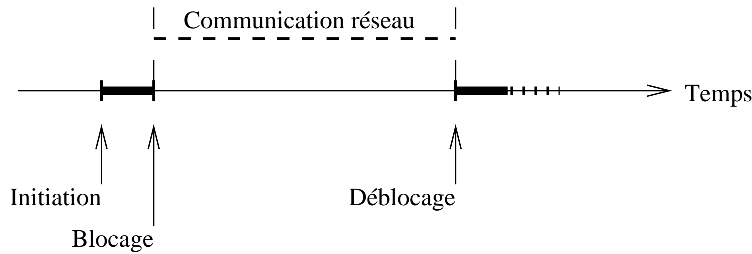

FIG. 6.2 – *Communication d'un processus lourd.*

Dans la figure 6.2, un processus lourd initie une communication, celle-ci se fait sur le réseau et dès qu'elle est finie, le processus est débloqué. Le "point de déblocage" coïncide avec le déblocage effectif car le processeur n'exécute pas d'autres calculs pendant la communication et est donc immédiatement disponible quand elle termine (on ne tient pas compte ici d'autres processus lourds qui pourraient éventuellement être exécutés pendant que le processus communicant est bloqué).

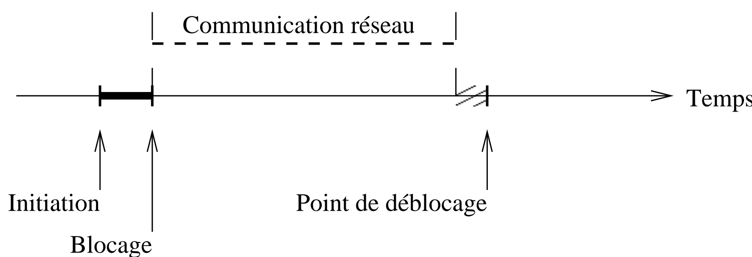

FIG. 6.3 – *Communication d'un processus léger.*

La figure 6.3 présente la communication d'un processus léger. On note que le "point de déblocage" se situe légèrement au delà de la terminaison de la communication. C'est la perte de réactivité due au fait que le processeur exécute (ou exécute potentiellement) d'autres calculs pendant que la communication se fait et ne prend pas immédiatement connaissance de la terminaison de celle-ci (le déblocage peut se situer au delà de ce point, si d'autres fils s'exécutent sur le processeur). Entre le moment où le fil communicant se bloque, et le moment où il est débloqué, d'autres fils prêts peuvent s'exécuter sur le processeur. Même si la performance "brute" de la communication est inférieure à celle d'un processus lourd monoprogrammé, le temps perdu peut être exploité par d'autres fils.

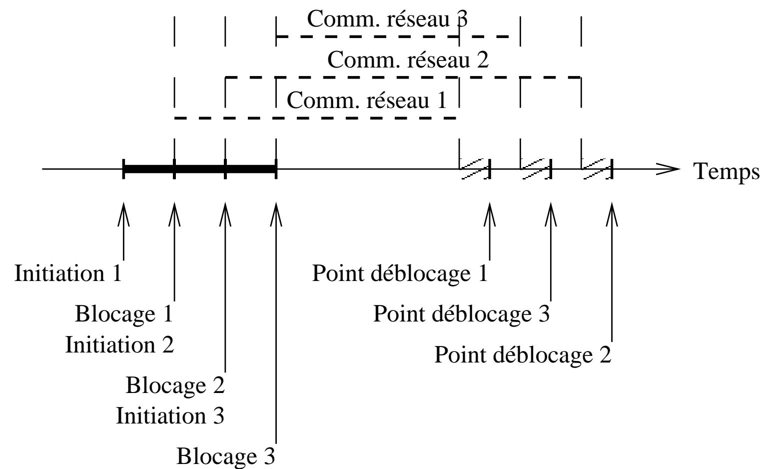

FIG. 6.4 – *Communication de plusieurs processus légers.*

Dans la figure 6.4, plusieurs processus légers effectuent des communications en même temps. On note toujours la perte de réactivité du système, mais également son pouvoir à recouvrir des opérations non conflictuelles : les communications sur le réseau progressent en parallèle (on suppose que le réseau utilisé le permet ou que les communications sont dans des phases n'entrant pas en conflit dans les couches logicielles et matérielles du réseau) et l'initiation de certaines communications se fait en même temps que la progression de celles faites antérieurement. Ici aussi quand les trois fils sont bloqués, le processeur est disponible pour l'exécution d'autres fils. La perte de réactivité sur chaque communication est amortie quand plusieurs communications sont faites en parallèle. La figure 6.4 présente un entrelacement particulier. En pratique, si d'autres fils ont des calculs à effectuer, ceux ci se feront pendant les communications des fils de la figure.

Sur les figures 6.3 et 6.4 ont été représentés les retards encourus par les processus légers à la terminaison des communications. La baisse éventuelle du débit ne l'a pas été. Voir section 6.4.2.

6.2 Objectifs

Comme énoncé en section 5.1, page 61, Athapascan-0b se veut **portable**, **efficace** et présentant les **fonctionnalités** nécessaires à une programmation parallèle aisée.

L'atteinte de ces objectifs est un compromis car ils ne sont pas indépendants les uns des autres. Une recherche poussée d'efficacité peut nuire à la portabilité. De même, certaines fonctionnalités sont incompatibles avec l'objectif d'efficacité, d'autres avec la portabilité. Par exemple, une fonctionnalité nécessitant la connaissance d'un état global sera difficile à rendre efficace. De même, une fonctionnalité ne pouvant être construite sur des couches logicielles standard sera difficilement portable.

Le compromis **idéal** entre ces trois objectifs n'existe pas. Le développement d'un

noyau exécutif tel qu'Athapascan-0b repose sur des choix et des décisions subjectives. Dans le cas d'Athapascan-0b, les critères d'efficacité et de portabilité ont été intégrés dans le processus de spécification de l'exécutif depuis les toutes premières étapes. Les fonctionnalités jugées nécessaires à la programmation parallèle ont pu facilement être intégrées dans Athapascan-0b car leur implantation n'entraînait ni perte d'efficacité du noyau ni l'utilisation de fonctionnalités non présentes dans les bibliothèques standard utilisées.[1]

6.3 Architecture

Le développement initial d'Athapascan-0b a été fait sur une machine IBM SP1, qui ne disposait à l'époque que de processus légers implantés en espace utilisateur. L'architecture d'Athapascan-0b a donc été adaptée à ce type de fonctionnement. Ensuite, des portages ont été réalisés vers des systèmes dans lesquels les processus légers étaient gérés par le noyau, et Athapascan-0b a été modifié pour en tirer partie. La description de l'architecture d'Athapascan-0b suit cette évolution et commence donc par une description de l'implantation originelle.

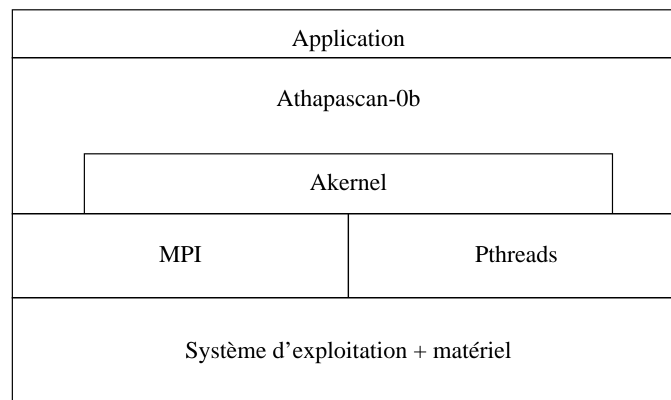

FIG. 6.5 – *Architecture d'Athapascan-0b.*

Athapascan-0b est structuré en deux couches. La couche basse s'appelle Athapascan-kernel ou Akernel, effectue l'intégration de MPI et des *Pthreads* et offre une interface *thread-aware*[2] pour la communication.

La couche haute est celle qu'utilise le programmeur d'application Athapascan-0b et qui est décrite au chapitre 5. La réalisation de cette couche s'appuie sur Akernel mais utilise aussi directement des fonctionnalités MPI et *Pthreads*. Elle n'a pas de nom

1. Pour être complètement honnête, il faut ajouter que les premières versions d'Athapascan-0b comportaient des fonctionnalités plus complexes de structuration du calcul en groupes qui n'étaient ni implantables efficacement ni très populaires parmi les utilisateurs d'Athapascan-0b. Il en a résulté la simplification ayant conduit à la présente version.

2. Voir section 3.7.1 page 42.

particulier, aussi l'appellera-t-on simplement Athapascan-0b. La figure 6.5 présente un schéma de l'achitecture d'Athapascan-0b.

6.3.1 Athapascan-0b

La couche supérieure est construite au dessus d'Akernel et n'a donc pas à s'occuper des problèmes de mariage des processus légers et des communications. Elle utilise l'interface d'échange de messages d'Akernel, qui est semblable à celle de MPI. Passons en revue les détails d'implantation des différentes fonctionnalités d'Athapascan-0b.

Initialisation et terminaison

L'initialisation des différentes tâches Athapascan-0b est une opération locale aux tâches qui n'implique pas de communications. Au démarrage de chaque tâche, un seul processus léger est en exécution et exécute la fonction `main` (démarrage normal d'un processus lourd). L'initialisation d'Athapascan-0b se fait en deux temps (deux appels), entre lesquels l'utilisateur peut effectuer des initialisation propres à son application. Le premier appel effectue les initialisations de bas niveau, comme le démarrage d'Akernel qui à son tour initialise MPI et les processus légers ainsi que la mise en place des structures de données nécessaires au fonctionnement d'Athapascan-0b. En fin de la première phase, l'utilisateur peut effectuer des initialisations de son application comme la déclaration des services et l'initialisation des bibliothèques par exemple. Dans la deuxième phase de son initialisation, Athapascan-0b crée les démons dont il a besoin.

Quand les initialisations sur chaque tâche sont terminées l'exécution de la fonction `main` continue, et l'application prend alors le contrôle des évènements.

L'application parallèle dans son ensemble se termine quand les fonctions `main` de **toutes** les tâches ont appelé la fonction de terminaison. L'application de l'utilisateur a la charge de terminer ses communications en cours (envois non encore reçus et réception non satisfaites) avant d'effectuer la terminaison. La terminaison n'est pas une opération locale, puisqu'elle implique une barrière entre toutes les tâches. Quand cette barrière est finie, chaque tâche arrête les démons qu'elle a démarré (à l'aide d'un code d'appel de service particulier) puis exécute le code de terminaison d'Akernel qui se charge à son tour de terminer son démon et d'appeler la terminaison de MPI.

Ports

Les ports et leurs étiquettes Athapascan-0b associées sont construits au dessus des étiquettes et des communicateurs MPI. Deux types de ports existent en Athapascan-0b : les ports locaux et les ports globaux. Un port local est créé localement par un fil d'une tâche et il est garanti qu'aucun autre appel de création de port, sur cette tâche ou une autre, ne renverra un port identique. Les ports globaux doivent être créés sur toutes les tâches dans le même ordre, et les appels de création de chaque port sur l'ensemble des tâches renvoient le même port. Ceci permet de construire des ports globaux sans

communications. Quelle que soit la tâche ayant crée un port local, ce port peut être utilisé comme la destination de messages sur toute tâche.

Un port est un couple composé d'une valeur de base pour l'étiquette MPI (voir paragraphe suivant) et d'un communicateur MPI. Au démarrage de la machine sont créés des communicateurs : un nombre de communicateurs égal au nombre de tâches, pour que chacune dispose d'un communicateur différent pour l'allocation des ports locaux ainsi qu'un autre communicateur pour les allocations de ports globaux sur toutes les tâches. Chaque tâche possède un compteur pour les ports locaux et un pour les ports globaux. Le compteur des ports globaux a au démarrage de la machine la même valeur sur toutes les tâches. La création d'un port sur une tâche consiste alors à déterminer quel communicateur et quel compteur utiliser, selon que le port créé est global ou local, à recopier l'identité du communicateur dans le port, ainsi que la valeur du compteur, qui devient la valeur de base de l'étiquette du port.

Après chaque création de port, le compteur correspondant est augmenté du nombre d'étiquettes Athapascan-0b différentes pouvant être utilisées avec le port. La valeur de l'étiquette MPI effectivement utilisée pour une communication est la somme de la valeur de base de l'étiquette du port et de la valeur de l'étiquette Athapascan-0b (les valeurs possibles de cette étiquette sont de zéro au nombre maximal d'étiquettes pour ce port moins un).

Déclaration de services

Afin d'assurer une numérotation cohérente des services sur l'ensemble des tâches sans nécessiter de communications, les services sont déclarés et numérotés dans le même ordre sur toutes les tâches. Un même service a donc un numéro identique sur toutes les tâches. Cette déclaration dans le même ordre se fait naturellement car toutes les tâches exécutent le même code au démarrage. Les adresses des fonctions de service sont stockées dans des structures internes à Athapascan-0b pour pouvoir être retrouvées lors d'un appel.

Appel de service à distance

Pour répondre aux appels de service à distance, Athapascan-0b met en place des démons sur chaque tâche à l'écoute des appels de service. Ces démons écoutent des ports système globaux créés dans la phase d'initialisation d'Athapascan-0b et qui sont utilisés par la primitive d'appel de service. Cette primitive est en fait l'envoi d'un tampon contenant l'identification du service à appeler ainsi que ses paramètres.

Deux démons sont mis en place dans chaque tâche. L'un écoute les appels de service normaux, l'autres les appels urgents. Deux ports globaux sont donc créés, un pour chaque démon. L'allocation du tampon pour le passage des paramètres au service est faite par le démon. Celui-ci teste (*probe*) l'arrivée du message d'appel de service pour déterminer sa taille avant de faire la réception, afin d'allouer un tampon de taille suffisante. Le tampon est automatiquement libéré quand le fil termine son exécution.

Les démons Athapascan-0b sont complètement indépendants du démon Akernel (voir section 6.3.2).

Gestion des processus légers

La gestion de la création de fils d'exécution est faite de la même façon, que ce soit lors d'une création locale ou d'un appel de service. À chaque fil en exécution dans une tâche est associée une zone de données privées. Cette zone est allouée à la création et libérée automatiquement quand le fil termine. Tous les paramètres passés à un fil le sont à travers cette zone mémoire (paramètre passé à un esclave ou tampon à un service). Dans le cas où le système de trace est activé, dans cette zone est stocké un identificateur du fil pouvant être utilisé par le code de prise de trace.

Il est également possible de réutiliser les fils d'exécution ayant terminé leur exécution pour économiser le coût de création et de destruction d'un fil.[3] Dans ce cas, quand la fonction exécutée par le fil se termine le fil se bloque sur un sémaphore en attendant d'être réactivé par une "création" ultérieure de fil de même priorité. La signalisation habituelle associée aux processus légers (attente de terminaison et de renvoi de résultat, détachement (`detach`) d'un fil etc...) continue à être utilisable par l'application mais est reconstruite au dessus du mécanisme de réutilisation. La zone de données privées sert également à stocker des informations pour la réutilisation.

La zone de données privées est utilisée dans tous les cas, même si elle n'est pas nécessaire pour une exécution sans prise de trace ni réutilisation des processus légers (sa non utilisation impliquerait d'écrire un code de création de fil différent pour ce cas). Le surcoût dû à l'utilisation de ce mécanisme (voir section 7.2.1 page 94) pourrait donc être évité dans ce cas.

6.3.2 Akernel

Akernel présente une interface de programmation permettant à de multiples fils d'exécution d'envoyer et de recevoir des messages, tout en étant développé au dessus d'une bibliothèque de communication non conçue pour la multiprogrammation légère. Une bibliothèque MPI "normale", c'est-à-dire non adaptée aux processus légers, n'est d'une part pas réentrante et d'autre part non *thread aware*. Ces deux points signifient qu'il est interdit à plus d'un fil de s'exécuter dans la bibliothèque et qu'aucun appel à MPI ne doit être bloquant, sous peine de bloquer l'ensemble des fils d'exécution de la tâche. Les incompatibilités entre MPI et les processus légers auraient pu être plus importantes (voir section 3.7.2 page 43) mais ce standard a été rédigé de façon à ne pas compromettre une utilisation future de la multiprogrammation légère, même si les premières implantations n'en faisaient pas usage.

---

3. Le noyau de processus légers utilisé pour le premier développement d'Athapascan-0b sur le SP1 ne libérait pas la mémoire d'un fil ayant terminé son exécution. Ce problème nous a forcé à prendre le chemin de la réutilisation des processus légers.

Contraintes

La première contrainte concernant le nombre d'appels simultanés à la bibliothèque se résout simplement en plaçant les appels à l'intérieur d'une section critique. Le verrou associée à cette section critique est utilisé par Akernel pour ses appels à MPI mais également par la couche supérieure quand celle-ci utilise MPI directement.

L'impossibilité d'effectuer des appels bloquants à la bibliothèque impose de ne faire que des appels non bloquants et utiliser ceux-ci pour simuler les appels bloquants des fils d'exécution.

Enfin, la diversité des noyaux de processus légers au dessus desquels fonctionne Akernel (DCE et les diverses implantations de *Pthreads* qui ne suivent pas toujours complètement la norme) a rendu nécessaire la mise en place d'un mécanisme d'harmonisation des appels. Akernel offre un tel mécanisme et cache à la couche supérieure les particularismes de telle ou telle implantation de processus légers.

Fonctionnalités étendues

Quand un fil effectue une communication asynchrone et ne désire pas attendre ultérieurement la terminaison de celle-ci, il peut passer une requête nulle[4] en paramètre à la primitive de communication. La communication sera gérée comme les autres communications, à ceci près que dès qu'elle termine, les structures en mémoire qui lui sont associées sont libérées.

Akernel propose aussi les sémaphores pour la synchronisation entre les processus légers ainsi que des primitives de manipulation de requêtes non associées à des communications. Des primitives permettent de créer une requête dans l'état bloqué et de débloquer une requête précédemment créée. Un fil peut tester ou attendre la terminaison d'une requête de ce type. Son attente ne sera satisfaite que quand un autre fil aura débloqué la requête. Ces fonctionnalités permettent de généraliser l'attente et le test de fin de communication à d'autres événements.

Akernel propose également un mécanisme d'"appel en retour" (*callback*) sur les attentes de fin de communication ou les attentes de requêtes. Une fonction *callback* et ses paramètres sont passés lors de l'appel de communication asynchrone (où à la création d'une requête dans l'état bloqué) et quand la communication termine ou que la requête est débloquée, la fonction est appelée avec les paramètres. La fonction est appelée dès qu'Akernel prend connaissance de la fin de la communication ou du déblocage de la requête, même si le fil n'est pas débloqué immédiatement. Ce mécanisme permet d'associer des traitements à des fins de communications et est utilisé principalement pour la prise de trace d'Athapascan-0b et par l'extension Athapascan-0b Formats qui seront décrites en détail dans des documents ultérieurs.

4. La requête Akernel joue un rôle similaire à la requête Athapascan-0b décrite section 5.3 page 67.

Le démon d'Akernel

Plusieurs choix sont possibles pour simuler des appels bloquants à l'aide d'appels non bloquants. Le principe en est que la terminaison de chaque communication en cours est testée périodiquement et le fil ne continue son exécution qu'une fois le test indiquant que la communication est finie. Deux choix sont possibles ici : soit un fil en attente teste en boucle la terminaison de la communication, cédant périodiquement le processeur à d'autres fils (entre deux tests de la communication, quand son quantum de temps est épuisé, avec une périodicité variable etc…), soit le fil se bloque sur une variable de synchronisation, à charge pour un autre fil de le débloquer le moment venu.

La solution du fil testant la terminaison de sa communication n'est pas intéressante dans la mesure où la réactivité d'un tel système est très faible. En supposant que tous les fils bloqués testent leurs communications à tour de rôle, le temps qui s'écoule entre deux tests faits par un fil donné correspond au temps de test de tous les autres fils bloqués augmenté du temps de toutes les commutations de contexte nécessaires qui de plus induisent un surcoût important à l'exécution.

Akernel a donc retenu la solution adoptée par la plupart des autres systèmes de ce type, à savoir l'existence d'un fil démon spécialisé dont le rôle est de tester périodiquement la terminaison des communications et de débloquer les fils dont les communications terminent.

Une communication bloquante se passe alors en plusieurs temps. Quand le fil effectue l'appel de communication, une communication asynchrone est faite via MPI à la place de la communication synchrone et dans les structures du démon est ajoutée la requête retournée par MPI associée à une variable de synchronisation. Le fil effectuant la communication se bloque sur la variable de synchronisation. Périodiquement (voir section 6.4), le démon s'exécute et teste par des appels non bloquants à MPI si chacune des communications en cours a terminé. Quand une communication a terminé, le démon débloque le fil en attente à l'aide de la variable de synchronisation.

Akernel avait pour objectif premier de cacher à Athapascan-0b les détails de l'intégration des communications et des processus légers. Dans les systèmes plus récents dans lesquels les processus légers sont gérés par le noyau et permettent à un fil de se bloquer en communication sans paralyser toute la tâche, le rôle d'Akernel devient flou et notamment le démon perd de son utilité. Ceci est vrai, mais pas totalement. Si pour l'utilisation de requêtes non nulles (le fil communiquant testera ou attendra ultérieurement la fin de la communication) les appels directs au système remplissent correctement le rôle d'Akernel, il n'en est pas de même pour les requêtes nulles. MPI propose une telle fonctionnalité (exprimée de façon un peu différente), mais l'utilisation des fonctions *callback* ou la prise de trace de la terminaison de la communication deviennent impossibles. Donc même dans des systèmes *thread aware* il est nécessaire d'avoir un démon au niveau Akernel, celui ci jouant cependant un rôle moins important que dans le cas de processus légers en espace utilisateur et étant de ce fait moins critique pour les performances d'Athapascan-0b.

6.4 Avancement des communications

Si certaines parties d'Athapascan-0b et d'Akernel pourraient être optimisées afin de présenter de meilleures performances, le point le plus critique (et le plus intéressant) à considérer concerne la scrutation du réseau et l'avancement et la terminaison des communications en cours. Cet aspect est pris en charge par le système si les processus légers sont gérés par le noyau mais quand ceux-ci sont en espace utilisateur, une certaine liberté est possible. Dans les deux cas, la problématique est la même dans la mesure où il faut résoudre le problème de l'avancement des communications, problème traité soit par le système, soit par Athapascan-0b.

6.4.1 Utilisation monoprogrammée

Afin de comprendre les enjeux de l'avancement et de la terminaison des communications, il faut comprendre quel est le comportement de la bibliothèque de communications en environnement monoprogrammé. Prenons l'exemple de MPI-F, sur lequel a été développée la première version d'Athapascan-0b.

Quand le processus lourd utilisant MPI-F effectue un envoi synchrone de message, MPI-F entre dans une boucle active qui teste sans arrêt le réseau pour voir si celui-ci est prêt à recevoir de nouveaux paquets (ce comportement est compréhensible puisque l'adaptateur de communication du SP1 n'est utilisable que par un seul processus lourd par nœud, il n'y a donc pas en fonctionnement normal d'autres processus utilisateur en attente du processeur). Dès que possible, un nouveau paquet composant le message est envoyé sur le réseau. Le nombre de paquets nécessaire varie bien sur selon la taille totale du message à émettre. La réception synchrone de message se passe approximativement de la même manière, à ceci près qu'il y a une phase d'attente du premier paquet puis les paquets suivants sont retirés du réseau par une boucle active au fur et à mesure de leur arrivée. Ces envois et réceptions synchrones sont ceux offrant le meilleur débit et la meilleure latence puisque le processus lourd est entièrement dédié à la communication.

Lors d'un envoi asynchrone, le processus lourd continue à s'exécuter alors que la communication n'est pas finie voire n'a pas encore commencé. MPI-F doit alors faire progresser la communication en parallèle avec l'exécution du processus lourd. Pour ce faire, deux méthodes sont utilisées conjointement. Chaque fois que le processus lourd appelle une primitive de communication de MPI-F, ce dernier en profite pour faire avancer la ou les communications en cours. Pour les cas où le processus lourd ne fait pas d'appels à MPI-F pendant longtemps, des interruptions logicielles sont déclenchées périodiquement et font alors également avancer les communications. Ces interruptions périodiques (faites toutes les 400 millisecondes) ne sont pas d'une très grande efficacité. À chacune d'elles MPI-F fait “un” avancement des communications et test du réseau. Sachant qu'il faut par exemple cinq tels avancements pour envoyer un message de 8000 octets cet envoi prendrait 2 secondes, à comparer avec le temps d'envoi par une primitive synchrone qui est inférieur à la milliseconde.

6.4.2 Utilisation multiprogrammée

Un des objectifs d'avancement des communications en utilisation multiprogrammée légère est de se rapprocher le plus possible du mode de fonctionnement monoprogrammé, c'est à dire faire avancer rapidement les communications en cours (débit) et prendre connaissance le plus rapidement possible des messages arrivant (latence). La latence ne peut être inférieure à l'intervalle de temps séparant deux tests du réseau. Si les tests du réseau ne sont pas déclenchés par celui-ci sur l'arrivée d'un message, une faible latence implique des tests fréquents d'où un surcoût important pour les calculs (pendant que le processeur teste le réseau il ne peut pas effectuer de calculs). L'accroissement du débit ne génère pas nécessairement un surcoût car l'envoi d'un message à un débit élevé ne nécessite pas plus d'interactions avec le réseau qu'un envoi à faible débit, il faut seulement que les interactions soient plus rapprochées dans le temps.

MPI-F sur le SP1 ne propose pas une signalisation liée à l'arrivée des messages. Il est donc nécessaire de tester explicitement le réseau de façon périodique. Afin que MPI-F teste le réseau et fasse avancer les communications il suffit de l'appeler avec un test d'une communication en cours.

Il n'y a pas de solution idéale valable pour tous les cas de figure. L'efficacité d'une solution dépendra du type des communications effectuées par l'application. Il est possible de faire des scrutations du réseau (associées à l'avancement des communications) :

– À la demande de l'application par un appel spécifique,

– Périodiquement, avec une période fixée,

– Quand tous les fils d'exécution de l'application sont bloqués sur des attentes de communications,

– Quand l'application appelle la bibliothèque de communication.

Des problèmes techniques propres à l'implantation de MPI-F ne nous ont pas permis d'effectuer des scrutations périodiques à période fixée du réseau, car la primitive des processus légers permettant ce genre d'appels (attente bornée dans le temps sur une variable de condition) était perturbée par les interruptions périodiques de MPI-F. Les trois autres méthodes de scrutation ont été implantées.

Dans les environnements de programmation légère disposant de priorités, il est nécessaire que le démon d'Akernel s'exécute avec une priorité plus élevée que les fils utilisateur. En effet, le démon doit pouvoir s'exécuter (à la demande de l'application et/ou périodiquement) même si d'autres fils sont prêts à s'exécuter. Pour ne pas monopoliser le processeur et créer un état de famine, le démon se bloque sur une variable de condition pour laisser les autres fils s'exécuter. L'attente sur la variable de condition peut être limitée dans le temps (voir section 3.4.2 page 33). Le démon reprend son exécution quand le délai d'attente est écoulé ou quand la variable est signalée par un

autre fil. Quand l'application souhaite faire avancer les communications par le démon, elle signale cette variable et débloque le démon. Afin de réveiller le démon quand tous les fils utilisateur sont bloqués, un fil de réveil de la plus basse priorité possible est créé et exécute une boucle qui signale indéfiniment la variable de condition. Ce fil de réveil ne sera exécuté que quand tous les fils sont bloqués (sa priorité est inférieure), réveillera le démon et sera immédiatement préempté par lui.

Afin d'assurer un débit proche de celui d'un processus monoprogrammé, il est important que l'avancement des communications (associé à la scrutation du réseau) se fasse le plus rapidement possible. Quand des communications sont bloquées dans Athapascan-0b, il faut donc faire des appels à MPI-F aussi souvent et rapidement que possible. Cet aspect a été particulièrement optimisé dans le démon d'Akernel et a conduit à une nette amélioration par rapport aux premières versions de ce démon. La perte de débit due à un avancement des communications trop lent se traduit dans les courbes de temps de communication comme un surcoût par octet qu'induirait Athapascan-0b.

L'utilisation multiprogrammée d'une bibliothèque de communication avec la stratégie d'attente active qui est celle d'Athapascan-0b peut s'avérer plus efficace pour les communications que l'utilisation classique de la bibliothèque par un processus lourd monoprogrammé. Ce fut le cas de LAM/MPI utilisée sur Solaris. Les performances de communication des appels synchrones d'Athapascan-0b (utilisant LAM/MPI comme bibliothèque de communication) étaient meilleures que celles des appels synchrones de LAM/MPI faits par un processus monoprogrammé. La raison était que le processus lourd bloqué sur une communication cédait le processeur à un autre processus lourd et de ce fait retardait la fin de la communication. Une communication en attente active faite par le processus lourd au dessus de LAM/MPI (par un appel asynchrone et une boucle effectuant des tests jusqu'à terminaison de la communication) a rétabli l'ordre normal des performances[5], à savoir LAM/MPI plus performant qu'Athapascan-0b.

6.5 Validation des choix

Le choix de réalisation d'Athapascan-0b au dessus de bibliothèques standard d'une part, et sa division en deux couches, la couche basse Akernel et la couche supérieure Athapascan-0b[6], s'est trouvé pleinement justifié quand le premier prototype tournait sur le SP1 et qu'ont commencé les portages. Ces deux choix de réalisation faisaient d'une part qu'il était facile de "faire tourner" Athapascan-0b sur une nouvelle architecture, en recompilant les sources en utilisant le noyau *Pthreads* et la bibliothèque MPI disponibles, et d'autres part, une fois le portage fonctionnel réalisé, l'ensemble des optimisations de performance avait lieu dans Akernel, la couche supérieure n'étant

5. N'oublions pas qu'Athapascan-0b était construit au dessus de LAM/MPI et qu'il ajoute obligatoirement un certain surcoût à l'exécution…

6. Le nom "Athapascan-0b" sert à désigner l'ensemble des deux couches ou uniquement la couche supérieure, selon le contexte…

pas modifiée. Les portages non optimisés se sont fait très vite, le temps d'installer MPI et de régler d'éventuelles déviations du noyau de processus légers relativement à la norme POSIX. La plupart du temps, en moins d'une journée Athapascan-0b tournait sur une nouvelle architecture. La phase d'optimisation est bien sur plus longue, et n'a pas été faite sur toutes les architectures. Actuellement, Athapascan-0b a été porté sur les plates-formes suivantes :

- IBM SP1 ou SP2 sous AIX 3.2 avec les processus légers DCE et MPI-F. Ceci est la version originelle d'Athapascan-0b.
- IBM SP2 sous AIX 4.x avec les processus légers conformes à *Pthreads* du noyau de AIX 4.x et le MPI *thread-aware* d'IBM (non encore sorti, le laboratoire LMC-IMAG étant un site bêta).
- Un réseau de stations HP-9000 sous HP-UX 10.x, avec les processus légers HP-UX DCE et LAM/MPI.
- Un réseau de stations DEC Alpha sous OSF/1 4.x avec les processus légers *Pthreads* OSF/1 et LAM/MPI.
- Un réseau de stations SGI sous Irix 6.x, avec les processus légers *Pthreads* de Irix et LAM/MPI.
- Un réseau de stations sous Solaris 2.5 (architecture Sparc ou Intel) avec les processus légers *Pthreads* de Solaris et LAM/MPI.
- Un réseau de stations à architecture Intel sous Linux 2.x avec des processus légers *Pthreads* et LAM/MPI.
- Cray T3E avec le noyau de processus légers Marcel et une bibliothèque MPI développée par Cray.

Chapitre 7

Performance d'Athapascan-0b

7.1 Introduction

L'évaluation d'Athapascan-0b se fait en deux étapes. Dans la première, la performance des primitives de base du système est mesurée et comparée à la performance des mêmes primitives dans les bibliothèques standard sur lesquelles est construit Athapascan-0b. Cette comparaison permet de déterminer le surcoût de base induit par Athapascan-0b.

Dans un deuxième temps sont évalués des comportements et des fonctionnalités plus complexes d'Athapascan-0b. La performance de ces constructions est comparée à la performance des primitives de base utilisées.

Dans ce chapitre, les expériences ont été faites sur la machine parallèle IBM SP1/2 tournant sous AIX 3.2, Athapascan-0b étant développé au dessus de la bibliothèque de communications MPI-F et du noyau de processus légers DCE d'IBM. Certaines mesures ont également pu être faites avec une version bêta du futur produit MPI d'IBM, tournant sur AIX 4.2 et intégrant les processus légers. Dans les deux cas, l'utilisation des nœuds de la machine parallèle était exclusive, c'est à dire qu'une seule tâche Athapascan-0b était en exécution sur chaque nœud et aucun autre processus utilisateur (les processus système habituels étaient présents). La machine utilisée possède un *High-Performance Switch* (HPS) TB2 [88, 87] et des processeurs Power 1 à 66 Mhz.

7.2 Fonctionnalités de base

La performance des primitives de base conditionne celle de tout programme s'exécutant au dessus d'Athapascan-0b. Cette section évalue la performance des principales primitives d'Athapascan-0b, que celles-ci soient locales – ne faisant pas intervenir de communications – ou distantes.

Les primitives locales évaluées concernent la manipulation des processus légers et comprennent la création d'un fil d'exécution local (esclave) et la commutation de fil (`yield`).

Les primitives évaluées impliquant des communications sont l'envoi et la réception de messages, l'appel de service et l'appel de service urgent.

L'évaluation d'une primitive peut se faire dans un système non chargé, dans lequel seule la primitive s'exécute en dehors des fils d'exécution nécessaires au fonctionnement d'Athapascan-0b. La performance d'une primitive dans ces conditions est celle la plus à même d'être comparée à la performance d'une primitive correspondante dans les bibliothèques utilisées. Il est également possible d'évaluer la primitive en environnement chargé. La charge (de calcul ou de communication) peut être indépendante de la primitive mesurée ou encore être le fait de plusieurs exécutions simultanées de la primitive. Ce dernier cas permet de mettre en avant des recouvrements entre exécutions de la primitive si le temps d'exécution croît moins vite que le nombre de primitives exécutées simultanément.

La démarche adoptée dans cette section consiste à mesurer les temps d'exécution des primitives de base énumérées ci dessus dans un environnement non chargé.

7.2.1 Création locale d'un fil d'exécution

La création d'un fil d'exécution local (esclave) dans Athapascan-0b implique la création d'un fil d'exécution par la bibliothèque utilisée et l'association à ce fil d'informations de gestion propres à Athapascan-0b.

Afin de tracer les courbes présentées plus loin, un programme crée des fils d'exécution exécutant chacun la fonction vide. La durée de chaque création est mesurée par l'horloge à haute résolution du SP1 et une courbe est tracée, donnant le temps de création d'un fil en fonction du nombre de fils déjà présents dans la tâche. Le programme a été exécuté cent fois et les courbes présentées sont la moyenne de ces exécutions.

AIX 3.2

La figure 7.1 présente les temps de création des processus légers pour le noyau de processus légers DCE et pour Athapascan-0b au dessus de AIX 3.2. La bibliothèque DCE présente deux particularités que l'on retrouve ensuite dans Athapascan-0b. La première est que le temps de création d'un processus léger n'est pas constant, et croît avec le nombre de processus légers. Il n'y a pas de raisons techniques particulières pour justifier un tel comportement, qu'on ne retrouve d'ailleurs pas sur d'autres systèmes. La deuxième particularité est que la création de processus légers dans la bibliothèque DCE possède deux modes. En observant en détail la courbe des temps de création (la figure 7.2 présente un extrait de la courbe DCE de la figure 7.1) on remarque qu'il y a deux modes de création des processus légers, un lent et un rapide. Environ un processus sur trois est créé dans le mode lent. Ce comportement peut être expliqué par une politique d'allocation de ressources à l'intérieur de la bibliothèque où les ressources sont allouées en une fois pour trois créations de fils. Athapascan-0b utilisant la bibliothèque DCE pour ses processus légers, on y retrouve ces deux modes de création.

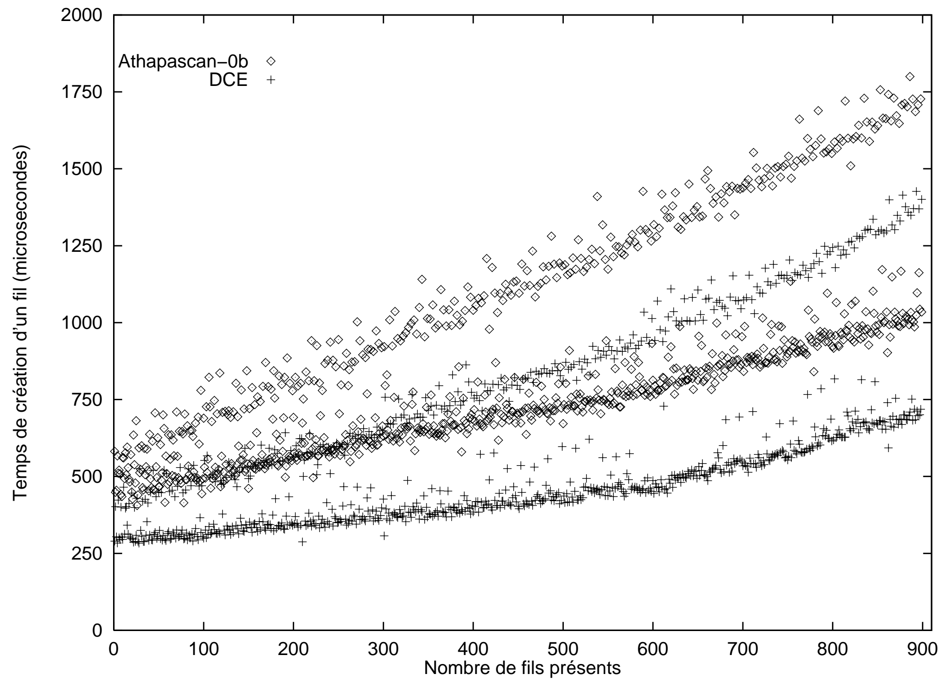

FIG. 7.1 – *Temps de création d'un processus léger DCE et Athapascan-0b sur AIX 3.2, fonction du nombre de processus légers déjà présents.*

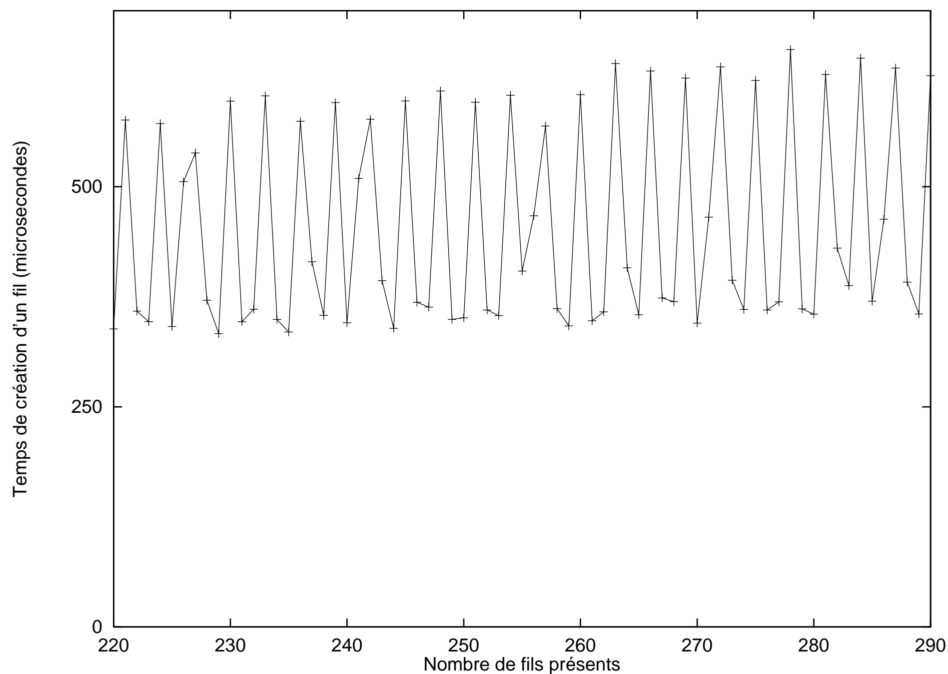

FIG. 7.2 – *Deux modes de création d'un processus léger DCE sur AIX 3.2.*

Le surcoût d'Athapascan-0b par rapport à la bibliothèque de processus légers est dû à la gestion par Athapascan-0b d'informations relatives aux processus légers (informations spécifiques à chaque processus léger), au mode de passage de paramètres à ceux-ci ainsi qu'à diverses opérations de synchronisation. Comme dit section 6.3.1 page 86, il aurait été possible dans certains cas de réduire ce surcoût. Cette optimisation est techniquement simple et améliorerait la performance de la création d'un processus léger dans ces cas.

Ce qui n'est pas expliqué par la gestion que fait Athapascan-0b des processus légers est la légère croissance du surcoût quand le nombre de processus légers augmente. L'implantation d'Athapascan-0b ne peut expliquer cette croissance (d'ailleurs sur AIX 4.2 on ne retrouve plus cette croissance du surcoût, voir titre suivant). Elle est à rapprocher de la croissance du temps de création d'un processus léger de la bibliothèque DCE. En effet, dans les actions que fait Athapascan-0b lors d'une création d'un esclave on trouve des appels à la bibliothèque DCE (utilisation de *mutex* pour protéger des accès à des données partagées, création de zones spécifiques à un fil (*thread-specific data*) etc. . .) et le coût de ces appels croît avec le nombre de processus légers existant sur le nœud.

AIX 4.2

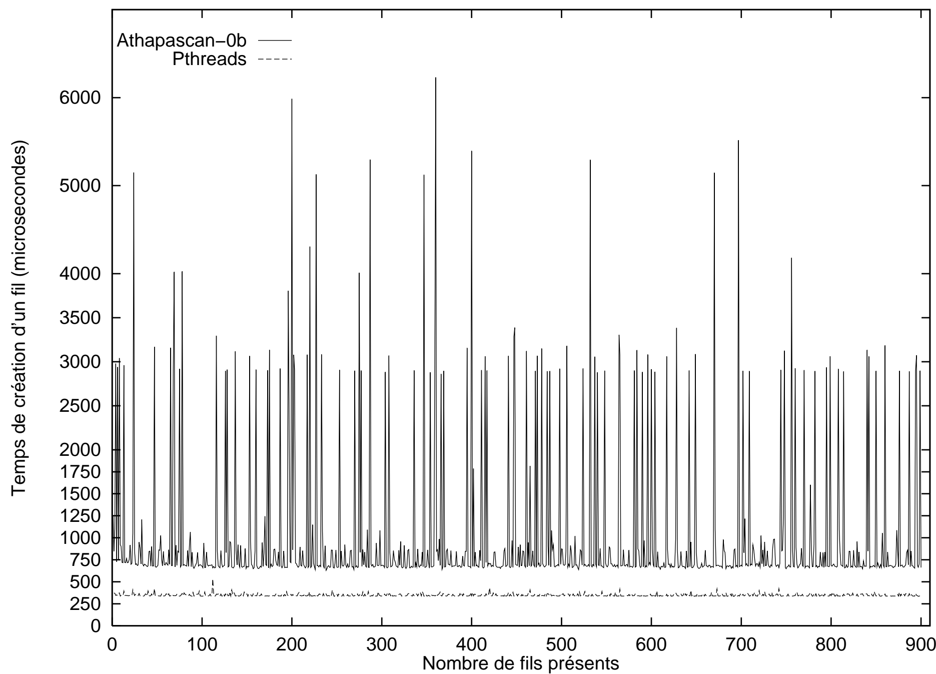

FIG. 7.3 – *Temps de création d'un processus léger Pthreads ou Athapascan-0b sur AIX 4.2, fonction du nombre de processus légers déjà présents.*

La figure 7.3 présente les temps de création de processus légers sur AIX 4.2. Le temps de création d'un processus léger AIX 4.2 est constant et ne dépend pas du nombre de processus légers déjà existants. On remarque alors que le temps de création d'un processus léger d'Athapascan-0b ne dépend pas du nombre de processus légers déjà présents dans le système et qu'il n'y a donc pas de croissance du surcoût (qui est de l'ordre de 330 microsecondes, quand on compare le coût de création par AIX 4.2 au coût de création par Athapascan-0b, “hors perturbation”).

L'ordonnancement des processus légers sur AIX 4.2 est de type temps partagé sans priorités, ce qui fait que tous les processus légers s'exécutent (ou essayent de s'exécuter s'ils sont bloqués) à tour de rôle. C'est l'explication avancée pour justifier les perturbations de la courbe d'Athapascan-0b, où le démon d'Akernel et les démons d'Athapascan-0b s'exécutent de temps en temps. On retrouve sur la courbe de temps de création des *Pthreads* de AIX 4.2 les mêmes perturbations à plus petite échelle. La figure 7.4 présente les temps de création des processus légers *Pthreads* sur AIX 4.2. Ici la perturbation est moindre (attention, l'origine des axes est hors de la figure) qu'avec Athapascan-0b sûrement du fait qu'il n'y a aucun processus léger prêt à s'exécuter mis à part les processus légers créés pour exécuter la fonction vide.

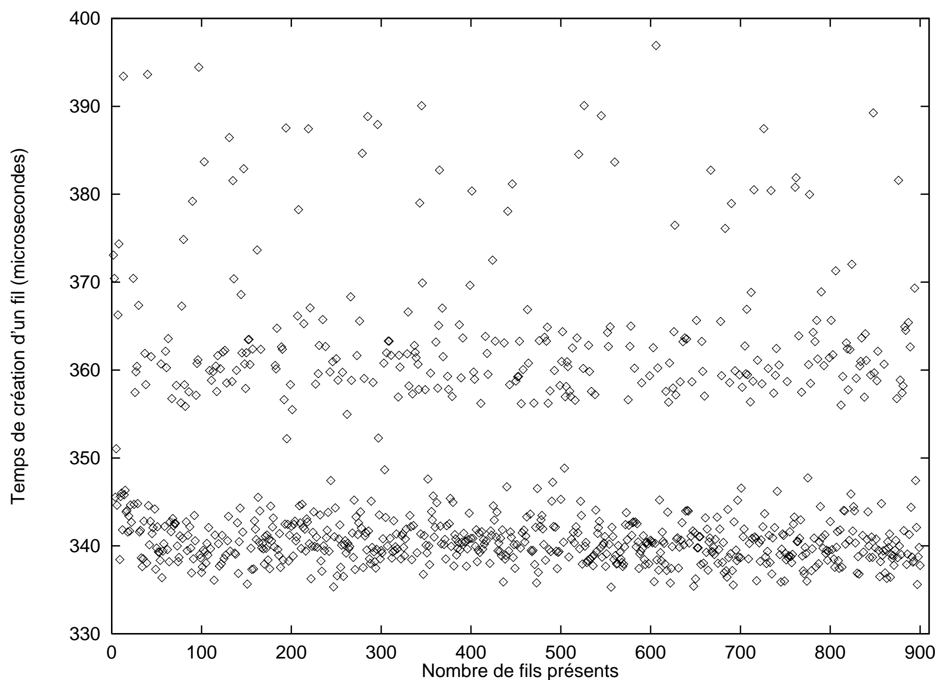

FIG. 7.4 – *Perturbation du temps de création d'un processus léger Pthreads sur AIX 4.2.*

7.2.2 Commutation de processus léger

Une commutation de processus léger a lieu quand un fil cesse son exécution sur un processeur et un autre fil reprend la sienne. Le temps d'une commutation de fil est le délai entre la fin de la dernière instruction exécutée par le premier fil et le début de la première instruction exécutée par le second.

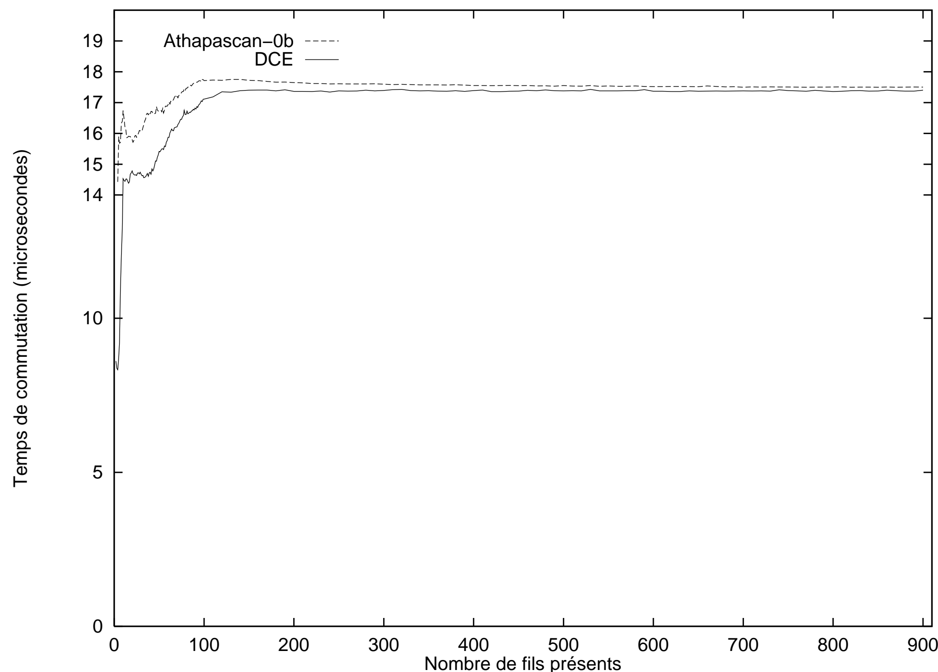

FIG. 7.5 – *Temps de commutation de fils DCE ou Athapascan-0b sur AIX 3.2, fonction du nombre de processus légers présents. Ordonnancement à partage de temps sans priorités.*

On mesure le temps de commutation moyen quand n processus légers sont présents sur le nœud. Chacun des processus légers exécute une boucle de `yield` (instruction qui cède le processeur à un autre processus léger). Quand tous les processus légers ont terminé leur boucle, $n \times$ nombre d'itérations de la boucle commutations ont été effectuées au total. Il suffit de diviser le temps d'exécution par cette valeur pour obtenir le temps de commutation moyen. On répète cette expérience pour différentes valeurs de n afin de tracer les courbes présentées dans cette section (toutes les courbes ont été obtenues avec 1000 itérations de la boucle).

AIX 3.2

La figure 7.5 présente les courbes obtenues sur AIX 3.2 à l'aide des processus légers DCE et Athapascan-0b. L'ordonnancement pour ces courbes a été le mode *other* des processus légers, qui correspond à un ordonnancement à partage de temps (*round-robin*) sans priorités.

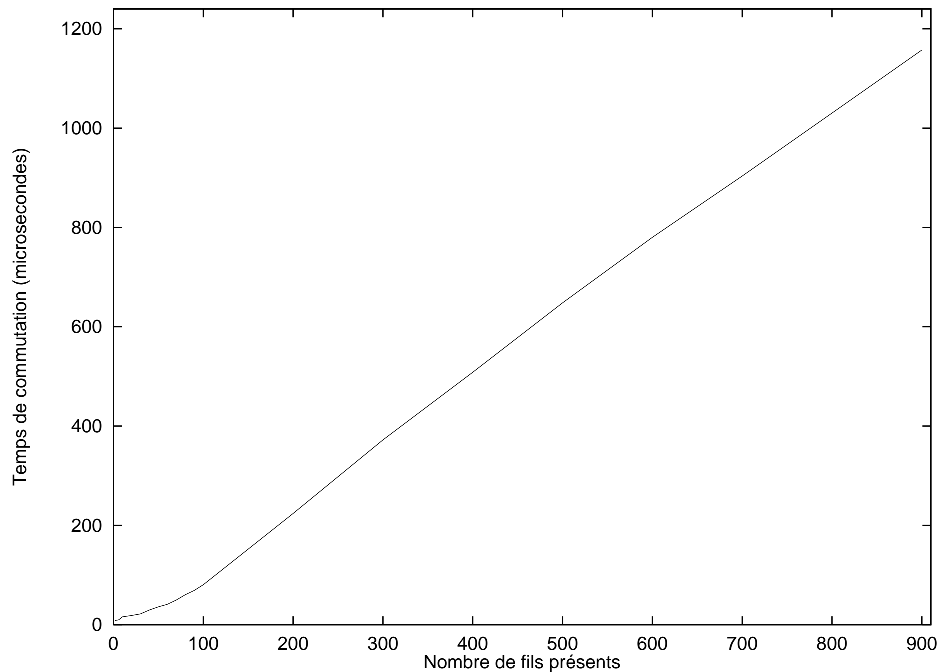

FIG. 7.6 – *Temps de commutation de fil DCE ou Athapascan, fonction du nombre de threads présents. Ordonnancement FIFO ou round-robin avec priorités.*

On constate que pour la bibliothèque DCE, au delà d'un certain nombre de processus légers, le temps moyen de commutation est constant. La courbe d'Athapascan-0b se comporte globalement de la même manière, présentant un écart par rapport à la courbe DCE qui tend à s'estomper quand le nombre de processus légers croît. Cet écart peut être dû à la présence des démons d'Akernel et d'Athapascan-0b qui sont périodiquement ordonnancés car le code d'Athapascan-0b n'ajoute aucun surcoût à la commutation de processus léger (il n'y a même pas d'appel de fonction, c'est directement l'appel de la bibliothèque de processus légers qui est fait à l'aide d'une macro).

Une autre explication de cet écart faisait intervenir les interruptions périodiques faites par MPI-F, qui "vole" ainsi des cycles processeur aux fils en exécution. Cette explication a été rejetée car on obtient une courbe similaire à celle de la figure en diminuant le nombre de `yield` faits par chaque processus léger, qui est de 1000 sur la courbe de la figure 7.5. Or, MPI-F générant ses interruptions toutes les 400 millisecondes n'aurait pas interféré de la même manière quand le nombre d'itérations de la boucle varie. De plus, les interruptions étant générées de façon relativement espacée (par rapport aux temps en jeu ici), une influence sur l'exécution se serait manifestée par une perturbation en marches d'escalier.

Le pic au début de la courbe d'Athapascan-0b n'a pu être expliqué par l'implantation d'Athapascan-0b et est surement à rapprocher du comportement légèrement chaotique des fils DCE dans cette zone.

Quand on utilise un ordonnancement avec priorité (FIFO ou *round-robin*) le temps

de commutation croît quasi linéairement avec le nombre de processus légers présents sur le nœud (expérience faite avec des processus légers de priorité identique). La figure 7.6 donne le temps moyen d'une commutation en fonction du nombre de fils présents. Cette courbe est obtenue avec des processus légers ordonnancés en FIFO ou *round-robin*, les résultats étant identiques. Ce comportement est un bogue de la bibliothèque de processus légers DCE utilisée et fait que jusqu'à 12 processus légers Athapascan-0b sur un nœud les ordonnancements FIFO et *round-robin* sont plus performants, au delà c'est l'ordonnancement *other* .

AIX 4.2

Il n'a pas été possible de faire des mesures précises du temps de commutation de fil sur AIX 4.2 du fait de l'instabilité du système utilisé (problème de libération de mémoire). Le comportement paraît néanmoins semblable à celui rencontré sur AIX 3.2 (courbe croissante à l'origine atteignant une valeur constante avant la centaine de processus légers) avec un temps de commutation légèrement plus grand à l'asymptote (de l'ordre de 18 microsecondes). Les mesures n'ont pas été assez précises pour déterminer la nature de l'écart entre la commutation dans Athapascan-0b et celle des *Pthreads* seuls.

7.2.3 Envoi et réception d'un message

Deux aspects de l'envoi et de la réception de messages doivent être évalués : la latence de l'envoi et son débit.

Pour s'affranchir des problèmes dûs à l'absence d'horloge globale,[1] la mesure du temps d'envoi et de réception se fait à l'aide d'un programme de "ping-pong" entre deux nœuds. Dans la version Athapascan-0b du programme, sur chacun des nœuds est créé un fil pour la communication. Sur un des nœuds le fil commence par effectuer un envoi puis attend en réception (nœud "ping") alors que sur l'autre nœud le fil commence par la réception et fait l'envoi ensuite (nœud "pong"). L'échange est répété un grand nombre de fois (5000 fois pour les courbes des messages de petite taille et 200 fois pour celles des messages de grande taille) et le temps total entre le premier envoi et la dernière réception est mesuré par le nœud "ping". Les courbes représentent le temps d'un aller retour divisé par deux, c'est à dire le temps qui s'écoule entre le moment où le fil "ping" commence l'envoi et le moment où le fil "pong" termine la réception correspondante (la durée d'un envoi de "ping" à "pong" est égale à la durée d'un envoi dans l'autre sens).

Les figures 7.7 et 7.8 présentent le temps d'un envoi et de la réception correspondante sur AIX 3.2, par la bibliothèque MPI-F et par Athapascan-0b. Sur la figure 7.7 on note le débit inférieur d'Athapascan-0b et sur la figure 7.8 la perte de réactivité par rapport à MPI-F.

1. Il se trouve que la machine SP1 sur laquelle ont été faites les mesures possède une horloge globale, mais notre démarche, tout comme Athapascan-0b, se veut portable.

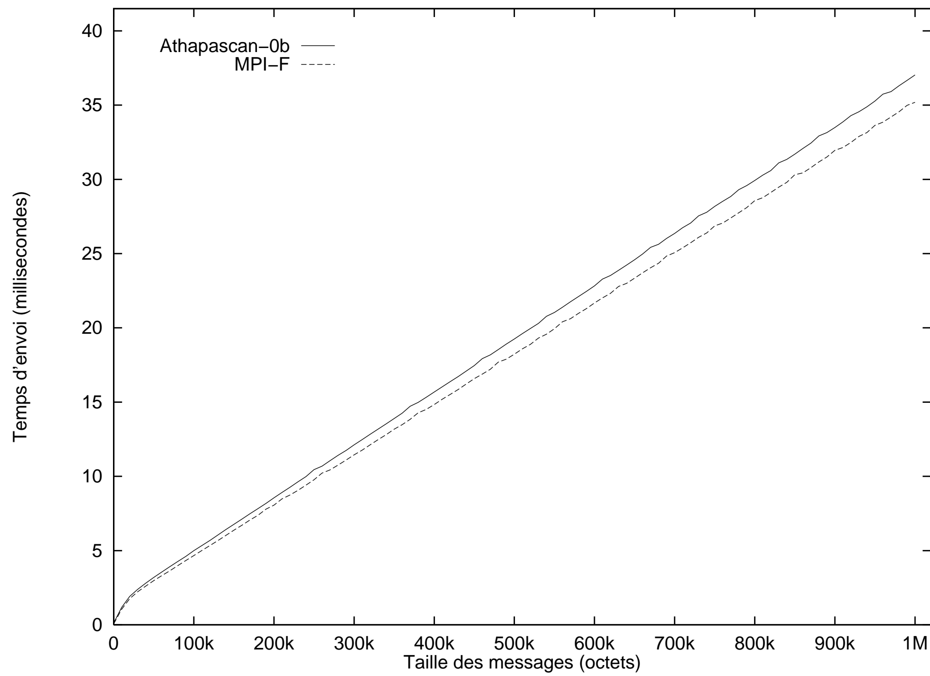

FIG. 7.7 – *Temps d'envoi de messages, Athapascan-0b et MPI-F sur AIX 3.2.*

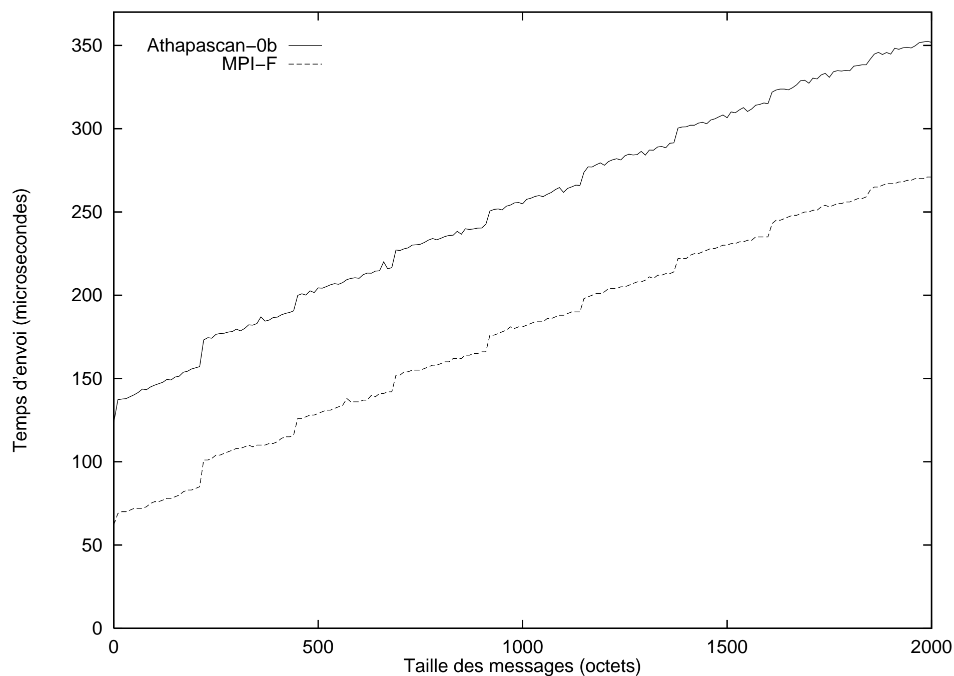

FIG. 7.8 – *Temps d'envoi de petits messages, Athapascan-0b et MPI-F sur AIX 3.2.*

La perte de débit, qui semble être un surcoût par octet qu'imposerait Athapascan-0b, est en fait dûe à l'avancement des communications, qui est fait moins rapidement par Athapascan-0b qu'il ne l'est par MPI-F (voir section 6.4.2 page 90 pour l'explication de ce phénomène). Cet aspect a été optimisé dans le démon d'Akernel, responsable de l'avancement des communications et les résultats présentés figure 7.7 sont bien meilleurs que ceux des premières versions d'Athapascan-0b. Un message d'un million d'octets est transmis par MPI-F en 35 millisecondes, ce qui représente un débit de $28,5$ millions d'octets par seconde ou $27,2$ Mo/s. Athapascan-0b transmet un même message en 37 millisecondes, pour un débit de 27 millions d'octets par seconde ou $25,7$ Mo/s. La perte de débit occasionnée par Athapascan-0b sur des gros messages est inférieure à 6%.

L'écart entre Athapascan-0b et MPI-F tel qu'il apparaît dans la figure 7.8 est moins facile à expliquer. La latence de MPI-F (envoi d'un message de taille nulle) est de 60 microsecondes et celle d'Athapascan-0b de 120 microsecondes (quand on passe à des messages de 10 octets, les temps d'envoi respectifs sont 69 et 137 microsecondes). L'exécution d'Athapascan-0b ayant généré cette courbe dispose du fil démon d'Akernel s'exécutant avec la plus haute priorité et avec un fil de réveil s'exécutant en plus basse priorité. Le fil de réveil ne s'exécute donc que quand tous les autres fils sont bloqués et réveille le démon bloqué en attente sur une variable de condition.

La première étape pour la compréhension de la performance d'Athapascan-0b consiste à savoir quel est le nombre d'appels nécessaires à MPI-F pour faire partir un message du nœud. En d'autres termes, est-ce que l'appel à une primitive de communication asynchrone de MPI-F fait par le fil utilisateur suffit à ce que le message dans son ensemble quitte le nœud ou est-ce que plusieurs appels sont nécessaires ? Une expérience a été faite (sans Athapascan-0b) dans laquelle un envoi de message asynchrone est suivi d'une boucle infinie de calculs sans appels à MPI-F. Sur le nœud destinataire du message on mesure le temps mis par le message à arriver (préalablement à l'envoi, les deux nœuds se sont synchronisés par une barrière). Le nœud émetteur n'effectuant plus d'appels à MPI-F, le seul avancement des communications se fait par les interruptions périodiques de MPI-F, qui ont lieu toutes les 400 millisecondes. Il est simple de déduire du délai de réception du message le nombre nécessaire d'appels à MPI-F (la durée d'un envoi sur le réseau est petite devant 400 millisecondes). Une vérification a ensuite été faite pour confirmer le nombre d'appels à MPI-F nécessaires pour chaque taille de message (l'envoi asynchrone du message est suivi d'un certain nombre d'appels à MPI-F pour faire avancer les communications puis de la boucle infinie de calcul). Le résultat de cette expérience est que les messages de taille inférieure ou égale à 3696 octets sont complètement envoyés dès l'appel de communication asynchrone[2].

Le code d'Akernel a ensuite été instrumenté, pour connaître de façon fine le déroulement des opérations lors de cette application de "ping-pong" et découvrir pourquoi cette opération est deux fois plus lente sur Athapascan-0b que sur MPI-F pour des

2. Nombre d'appels nécessaires à l'envoi d'un message : jusqu'à 3696 octets, un appel, de 3697 à 4096, 2 appels, de 4097 à 5552, 3 appels, de 5553 à 7408, 4 appels, de 7409 à 9264, 5 appels etc…

messages de taille nulle (qui correspondent au plus grand surcoût relatif d'Athapascan-0b).

Lors de l'envoi d'un message, les temps suivants ont été relevés :

– Entre l'appel fait par l'application à la primitive d'envoi d'Athapascan-0b et l'appel (asynchrone) fait par Athapascan-0b à MPI-F s'écoulent 7 microsecondes. Pendant ce temps, Athapascan-0b vérifie les paramètres de l'appel, effectue l'appel à la couche Akernel qui verrouille alors le *mutex* protégeant les appels à MPI-F et qui alloue les variables de synchronisation nécessaires pour que le fil puisse se bloquer sur la communication (*wait-context* en terminologie Athapascan-0b) et être réveillé ultérieurement. Ce n'est qu'ensuite qu'est fait l'appel à MPI-F (qui consomme 28 microsecondes, comprises dans les 60 microsecondes que met le message à arriver à destination).

– Suite au retour de l'appel à MPI-F, le *wait-context* est inséré dans les structures de données du démon d'Akernel et le *mutex* de protection des appels à MPI-F et de manipulation des structures internes du démon libéré. L'ensemble de ces opérations prend 5 microsecondes.

– Avant de se bloquer en attente de la terminaison de la communication (pour être débloqué par le démon d'Akernel), le fil effectue un test de la terminaison. Ce test lui indique que la communication est terminée (on a affaire à de petits messages qui sont complètement envoyés dès le premier appel à MPI-F). Le fil ne se bloque donc pas. Le temps total pris par le test est de 7 microsecondes (il y a nouvelle acquisition du *mutex* protégeant les appels à MPI-F).

– Une fois que la terminaison de l'appel est connue (que ce soit comme ici directement par le fil qui communique ou par le démon d'Akernel), le *wait-context* associé est marqué comme fini et sa synchronisation est débloquée. Ces opérations consomment 10 microsecondes.

– Une fois le déblocage effectué, il faut encore 5 microsecondes pour libérer les structures. Ceci est du au fait que la terminaison d'une communications est indépendante de l'attente de celle-ci par le fil (cette distinction est utile quand le démon d'Akernel débloque un fil). Ici les deux opération se suivent mais ne sont pas faites en une fois.

La réception du message se passe de façon similaire, à ceci près que sa terminaison n'est pas immédiate (de par la nature du programme de “ping-pong”, les réceptions sont postées en avance) :

– Comme pour l'envoi, 7 microsecondes s'écoulent entre l'appel à la primitive de réception d'Athapascan-0b et l'appel correspondant à MPI-F (qui consomme quant à lui 10 microsecondes).

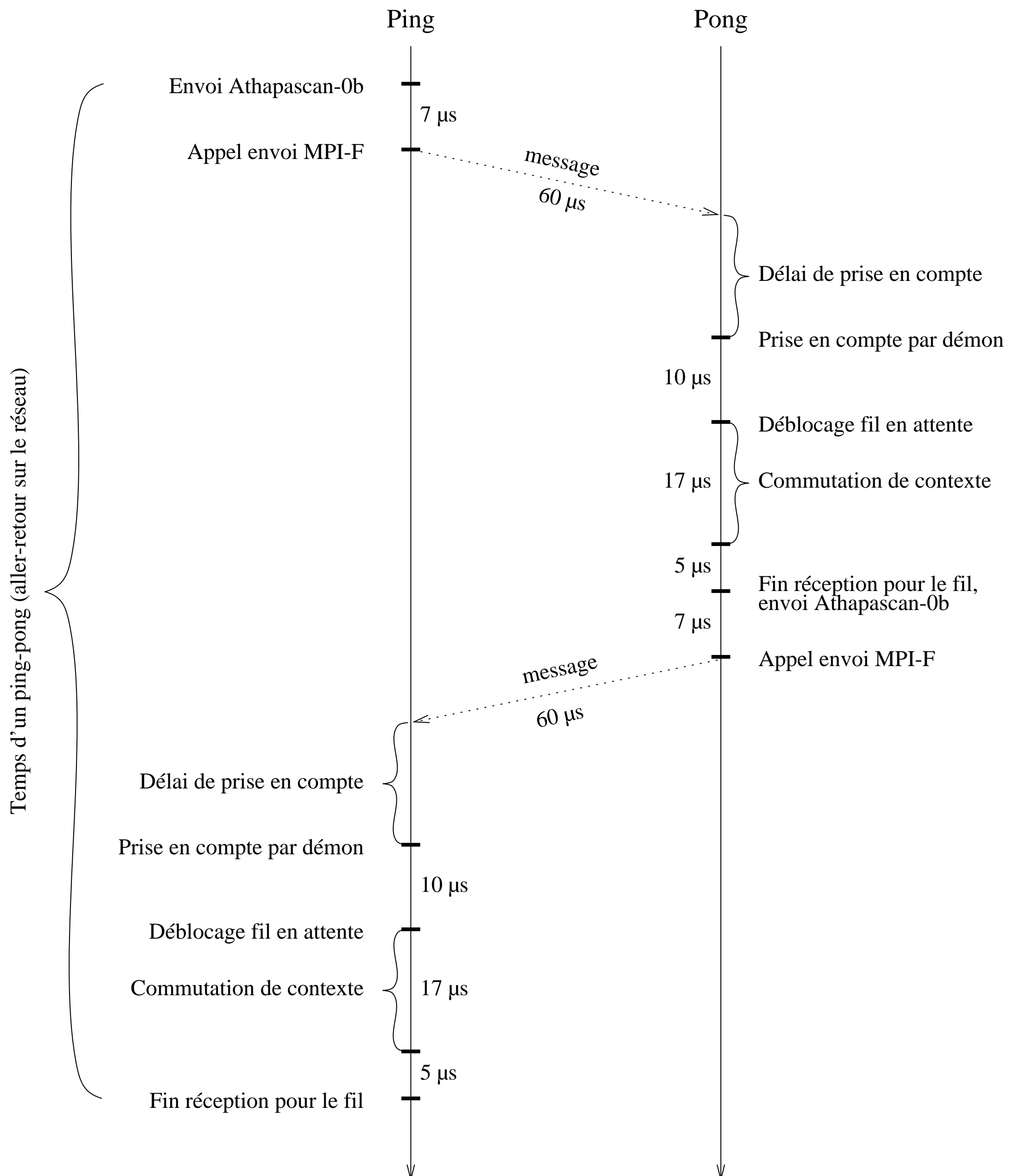

FIG. 7.9 – *Diagramme temporel d'un "ping-pong" pour messages de taille nulle en Athapascan-0b sur AIX 3.2.*

– Comme pour l'envoi, il y a également un délai de 5 microsecondes suite au retour de l'appel à MPI-F.

– Le fil se bloque ensuite en attente. Contrairement à l'envoi qui était fini dès le premier test, la réception dans le cas du “ping-pong” ne l'est pas, le fil se bloque et est réveillé par le démon.

– Quand le démon d'Akernel constate la fin de la réception, il débloque le fil par la synchronisation du *wait-context*. Entre le moment où le démon prend connaissance de la fin de la communication et le moment où il débloque le fil en attente s'écoulent 10 microsecondes.

– Pour que le fil débloqué reprenne son exécution il faut une commutation de contexte. Celle-ci prend 17 microsecondes.

– Le fil est réveillé dans l'appel d'attente de communication. La libération des diverses structures consomme 5 microsecondes.

On note dans le cas de la réception que l'opération de déblocage d'un fil dont la communication a terminé et la reprise de l'exécution de ce fil sont deux opérations distinctes. Les structures ne peuvent être libérées que quand le fil a terminé la synchronisation et a effectivement repris son exécution.

Les différentes étapes de l'émission et de la réception énumérées ci-dessus ne sont pas toutes sur le chemin critique de la mesure du temps d'un aller-retour (“ping-pong”) sur le réseau. La figure 7.9 présente un diagramme temporel d'un aller-retour de messages de taille nulle dans le programme du “ping-pong”. Sur cette figure ne sont représentées que les actions se trouvant sur le chemin critique de l'échange. Les délais sont ceux mesurés et expliqués dans les paragraphes précédents. Le délai d'envoi de message est celui de MPI-F utilisé seul.

Le “délai de prise en compte” apparaissant sur la figure correspond au temps qui s'écoule entre le moment où un message arrive sur un nœud et le moment où le démon d'Akernel prend connaissance que la réception correspondante doit terminer. Afin de déterminer ce délai moyen, lors de l'exécution du programme de “ping-pong” ont été mesurées toutes les dates auxquelles est testée la terminaison des réceptions (les réceptions dans ce programme sont postées à l'avance et ce n'est qu'après un certain délai qu'arrive le message correspondant. Le démon d'Akernel teste donc plusieurs fois une réception avant qu'elle ne soit satisfaite). Il apparaît que deux tests consécutifs de la même communication sont séparés d'environ 45 microsecondes. On peut en déduire que le délai moyen de prise en compte sera d'environ 22 microsecondes ($45/2$). Le délai moyen d'envoi de message serait donc de $7 + 60 + 22 + 10 + 17 + 5 = 121$ microsecondes, ce qui correspond bien aux valeurs mesurées (pour des messages de taille nulle).

Ce délai de 121 microsecondes peut être décomposé en trois parties. La première, de 60 microsecondes, est le délai de communication de MPI-F, indépendant d'Atha-

pascan-0b. La deuxième comporte les coûts de gestion des communications d'Athapascan-0b et sa durée est de $7 + 10 + 5 = 22$ microsecondes. Il est possible de réduire ce surcoût par une optimisation du code d'Athapascan-0b (voir paragraphe suivant). La troisième enfin ($17 + 22 = 39$ microsecondes) est en rapport avec la réactivité d'Athapascan-0b, et regroupe le délai de prise en compte du message et le temps de la commutation de contexte vers le fil communiquant, indépendant d'Athapascan-0b. L'existence du "délai de prise en compte", est intimement liée au mode de fonctionnement du réseau, dans lequel les terminaisons de communications doivent être explicitement testées par l'application (scrutation).

Si l'enchaînement des opérations faites par Athapascan-0b pour l'envoi de messages n'implique pas d'inutiles commutations de fils, l'efficacité des opérations en jeu pourrait être améliorée dans certains cas. Dans l'implantation actuelle, toute communication est référencée dans les structures du démon d'Akernel et des structures de synchronisation sont créées pour que la terminaison de la communication puisse être portée à la connaissance du fil communicant. Or, dans le cas de communications courtes, dans lesquelles la communication termine dès son envoi (comme c'est le cas dans l'exemple pour des petits messages), les mécanismes mis en jeu s'avèrent inutiles. Une simplification de la gestion des communications dans ce cas particulier réduirait notablement la latence d'Athapascan-0b. De même, alors qu'actuellement les communications synchrones sont implantées sous la forme d'une communication asynchrone suivi d'une attente, une implantation plus directe améliorerait les performances.

7.2.4 Appels de service

La mesure du temps nécessaire à un appel de service se fait de façon similaire à la mesure du temps d'envoi et de réception d'un message (section 7.2.3 page 100), pour compenser l'absence d'horloge globale. Par mesure de commodité, on parlera ici aussi de nœud "ping" et de nœud "pong", même si le programme d'appel de service n'est pas à proprement parler un "ping-pong".

Pour mesurer avec précision le temps d'un appel de service, le nœud "ping" appelle un service sur le nœud "pong", qui a son tour appelle un service sur le nœud "ping" qui rappelle le service du nœud "pong" etc... Ces services s'appellent mutuellement un certain nombre de fois, et on mesure le temps total pris par cette exécution. La prise de temps se fait sur le nœud "ping" qui effectue le premier appel de service et sur lequel s'exécute le dernier service appelé. Plus précisément, le premier appel de service est fait par le fil principal du nœud "ping" qui préalablement a fait une prise de temps et qui se bloque ensuite sur un sémaphore. Le service du nœud "ping" compte le nombre d'appels déjà faits. Tant que ce nombre n'a pas atteint le nombre total d'appels à faire, il refait un appel au service du nœud "pong". Quand le nombre total d'appels est atteint, le service signale le sémaphore. Le fil principal débloqué refait une prise de temps et connaît donc la durée totale de l'échange. Les deux courbes supérieures de la figure 7.10 représentent le temps d'un appel de service. Chaque point de ces courbes est le temps moyen d'un appel de service parmi 800 (400 dans chaque sens).

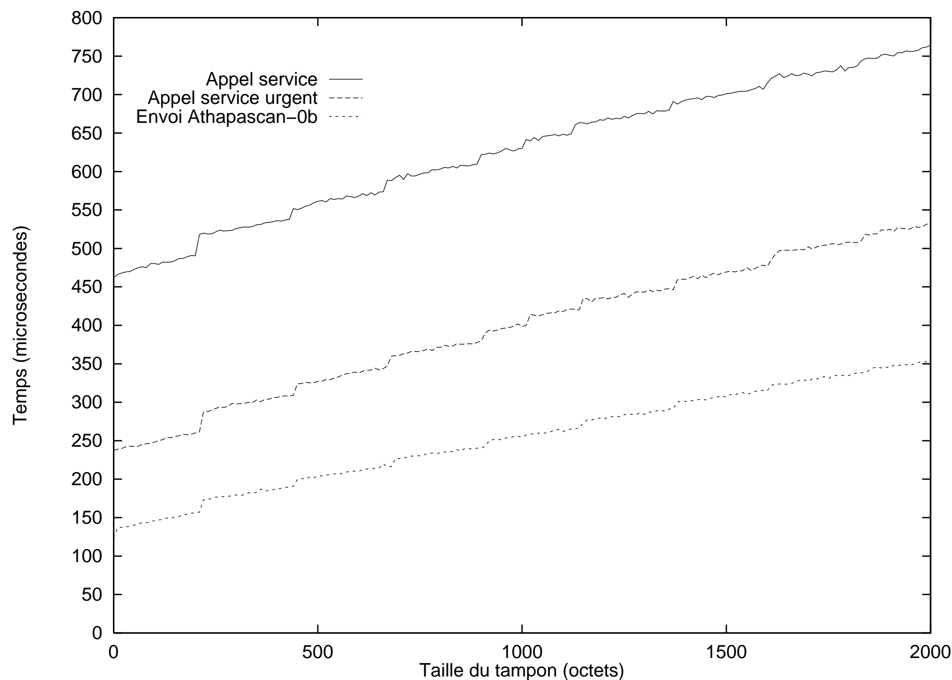

FIG. 7.10 – *Temps d'appel de service, de service urgent et d'envoi. Athapascan-0b sur AIX 3.2.*

L'appel d'un service est bien sur plus lent que l'envoi d'un message, car il implique une gestion du tampon des paramètres et de l'appel de la fonction sur le site distant, ainsi qu'une création de processus léger dans le cas de l'appel non urgent. On note sur la figure 7.10 que le surcoût des deux formes d'appel de service sur l'envoi d'Athapascan-0b est légèrement croissant. Ce phénomène n'a pas pu être expliqué. Pour de gros messages (un million d'octets), ce surcoût est de l'ordre de 500 microsecondes pour l'appel de service urgent et de 600 microsecondes pour l'appel de service avec création de fil. Il n'a également pas été trouvé d'explication satisfaisante au rapprochement des performances des deux types d'appels de service.

L'appel de service et l'appel de service urgent d'Athapascan-0b sont traités par des fils spécialisés (démons d'Athapascan-0b), au dessus de la couche d'échange de messages d'Akernel. Leur performance s'en ressent, et un traitement au niveau du démon d'Akernel permettrait une efficacité supérieure en éliminant certaines des étapes du décodage de l'arrivée d'un appel de service. Même pour les services normaux (non urgents), un décodage de l'appel et une création du fil exécutant le service directement par le démon d'Akernel permettrait de gagner les temps de déblocage et de commutation vers le démon d'Athapascan-0b gérant les appels de service.

7.3 Fonctionnalités composées

Dans cette section, nous allons voir le comportement d'Athapascan-0b face à des programmes plus complexes que précédemment. Ces programmes combinent différentes fonctionnalités d'Athapascan-0b. Ce n'est plus une fonctionnalité isolée qui est étudiée, mais le fonctionnement global d'un programme, avec les interactions qui peuvent avoir lieu entre ses différentes parties (exécutées ou pas dans des fils séparés).

7.3.1 Recouvrements

Nous allons montrer ici qu'Athapascan-0b effectue des recouvrements des communications entre elles, et des communications avec des calculs.

Recouvrement communications

Cette expérience est très semblable au "ping-pong" de la section 7.2.3 page 100.[3] Si précédemment deux nœuds échangeaient des messages par l'intermédiaire de deux fils (un sur chaque nœud), il y a ici plus d'un fil communicant par nœud, et donc autant de "ping-pongs" en parallèle entre des fils de deux nœuds (chaque fil de chaque nœud ne communique qu'avec un seul fil de l'autre nœud). Les figures 7.11 et 7.12 présentent le temps total de l'échange "ping-pong", avec divers nombres de fils communicants (les courbes tracées représentent le temps total de l'échange de l'ensemble des fils).

Le diagramme temporel de la figure 7.9 et les explications afférentes permettent de déterminer la durée incompressible d'une communication, c'est-à-dire le temps minimum que prend une nouvelle communication, quand le recouvrement est à son maximum. Cette durée incompressible est majorée par la durée pendant laquelle le processeur de calcul s'occupe de la communication. En effet, deux communications successives doivent être gérées l'une après l'autre par le processeur de calcul.

La durée d'utilisation du processeur de calcul pour un envoi de taille nulle est la suivante (en microsecondes) : 7 entre l'appel d'Athapascan-0b et l'appel MPI-F correspondant, 28 pour l'appel MPI-F proprement dit, 5 pour la mise à jour des structures d'Akernel, 7 pour le test de terminaison fait par le fil, 10 pour le déblocage de la synchronisation du fil et 5 pour la libération des structures (tous ces temps sont détaillés dans les explications de la figure 7.9, mais n'apparaissent pas tous sur la figure). Au total, lors d'un envoi, le processeur de calcul est utilisé pendant $7+28+5+7+10+5 = 62$ microsecondes. Ceci signifie qu'avec le meilleur recouvrement possible, chaque nouvelle communication prendrait au moins 62 microsecondes.

À partir des courbes de la figure 7.11, on détermine le coût additionnel de chaque nouveau fil communiquant. Ce délai est égal à la durée de communication de ce fil non recouverte par les communications des autres fils. On note ainsi que pour des messages de taille nulle, si un fil communique en 120 microsecondes, deux fils n'en nécessitent

3. D'ailleurs, le même programme de test a été utilisé dans les deux cas, il permet de choisir le nombre de fils communicants.

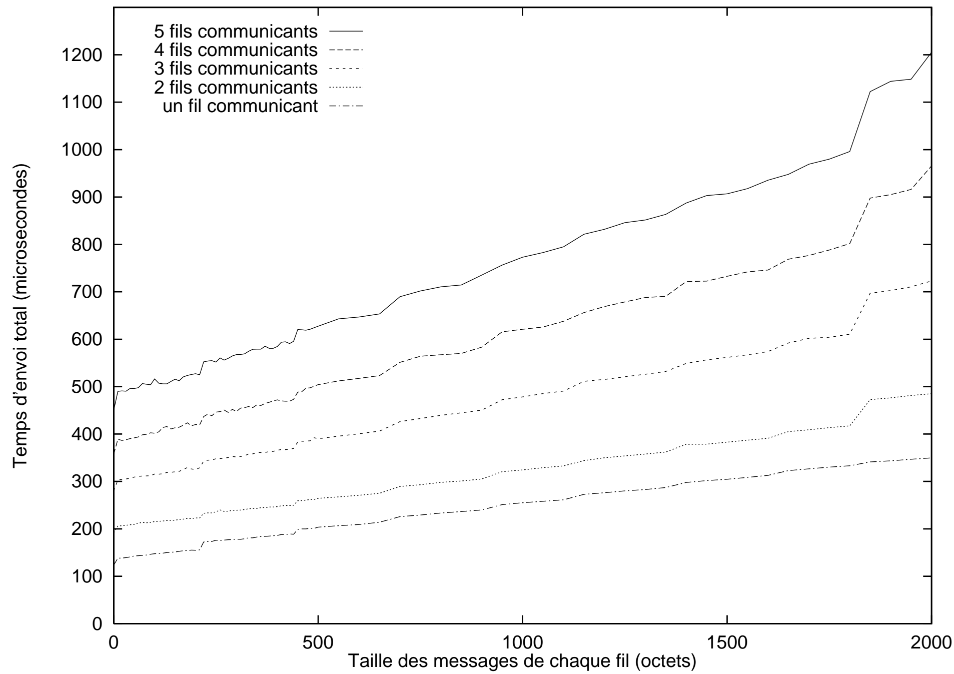

FIG. 7.11 – *Temps total d'un échange "ping-pong" avec plusieurs fils communicants par nœud. Athapascan-0b sur AIX 3.2.*

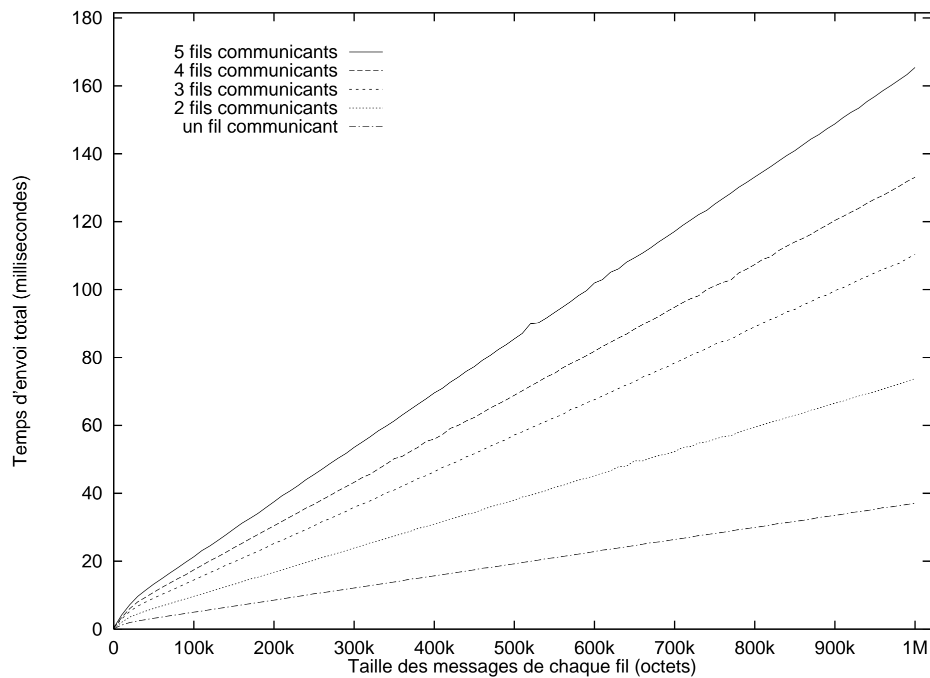

FIG. 7.12 – *Temps total d'un échange "ping-pong" avec plusieurs fils communicants par nœud – gros messages. Athapascan-0b sur AIX 3.2.*

que 190. Le deuxième fil n'ajoute donc que 70 microsecondes au temps d'exécution, les 50 microsecondes restantes (120 microsecondes est le temps d'une communication de taille nulle pour un fil seul) se superposent (sont recouvertes) avec les opérations d'envoi du premier fil. Le troisième fil rajoute quant à lui 90 microsecondes aux deux premiers, le quatrième 80 et le cinquième 90. Ces durées sont toutes compatibles avec la durée minimale d'une nouvelle communication, qui comme on l'a vu est égale à 62 microsecondes (mais peut être supérieure, du fait de l'activité additionnelle du démon d'Akernel et des commutations de contexte entre fils).

Dans la figure 7.12, c'est le débit du réseau qui limite le recouvrement des communications. On note cependant que le débit cumulé des fils communicants d'Athapascan-0b ne diminue pas (pas de perte d'efficacité due à la multiprogrammation légère). Au contraire même, le débit cumulé augmente. Avec des messages de taille un million d'octets, un fil Athapascan-0b seul communique à 27 millions d'octets par seconde et MPI-F à $28,5$ millions d'octets par seconde (voir section 7.2.3 page 100). Quand le nombre de fils communicants dans Athapascan-0b est supérieur, les débits cumulés sont les suivants (en millions d'octets par seconde) : $27,4$ pour 2 fils, $27,3$ pour 3 fils, 30 pour 4 et 5 fils. Non seulement un grand nombre de fils utilise le réseau de façon plus efficace, mais le débit cumulé ainsi atteint est supérieur à celui obtenu avec MPI-F en utilisant des communications bloquantes.

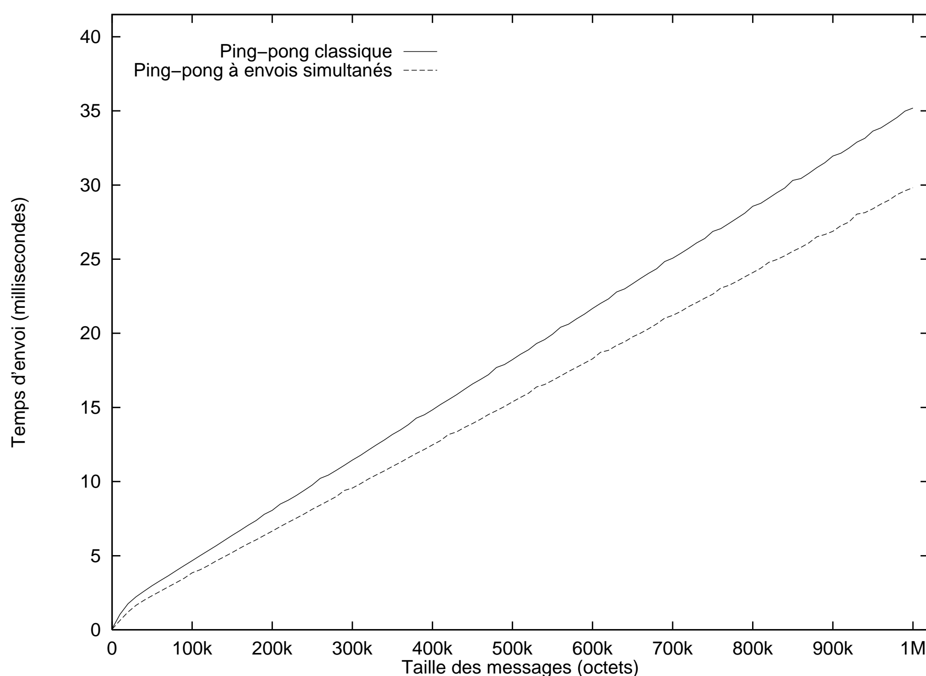

FIG. 7.13 – *"Ping-pong" classique et "ping-pong" dans lequel les messages sont envoyés simultanément des deux nœuds. MPI-F sur AIX 3.2.*

Mais la comparaison du débit cumulé d'un programme de "ping-pong" avec multiprogrammation légère (en Athapascan-0b) avec le débit d'un programme de "ping-

pong" dans lequel les communications sont bloquantes n'est pas complètement honnête. En effet, dans un "ping-pong" à communications bloquantes, à tout moment il n'y a qu'un seul message en transit sur le réseau, et les messages ne peuvent pas se croiser (il y a un lien de causalité entre l'arrivée d'un message sur un nœud et le départ d'un message de ce nœud). Or, dans un "ping-pong" à multiples fils d'exécution, rien n'empêche les échanges de différents fils de se croiser sur le réseau. Afin de faire une comparaison équitable, un programme de "ping-pong" a été écrit sur MPI-F dans lequel les rôles joués par les deux nœuds sont identiques, chaque nœud effectuant un envoi asynchrone suivi d'une réception asynchrone puis se bloque en attente de la terminaison de ces deux communications. La courbe représentant la durée d'un tel échange, comparé à la durée du "ping-pong" classique, est présentée dans la figure 7.13. Le débit pour des messages de taille un million d'octets est alors de $33,4$ millions d'octets par seconde (contre $28,5$ pour le "ping-pong" classique). La performance d'Athapascan-0b en "ping-pong" à multiples fils d'exécution paraît plus raisonnable.

Recouvrement calcul et communications

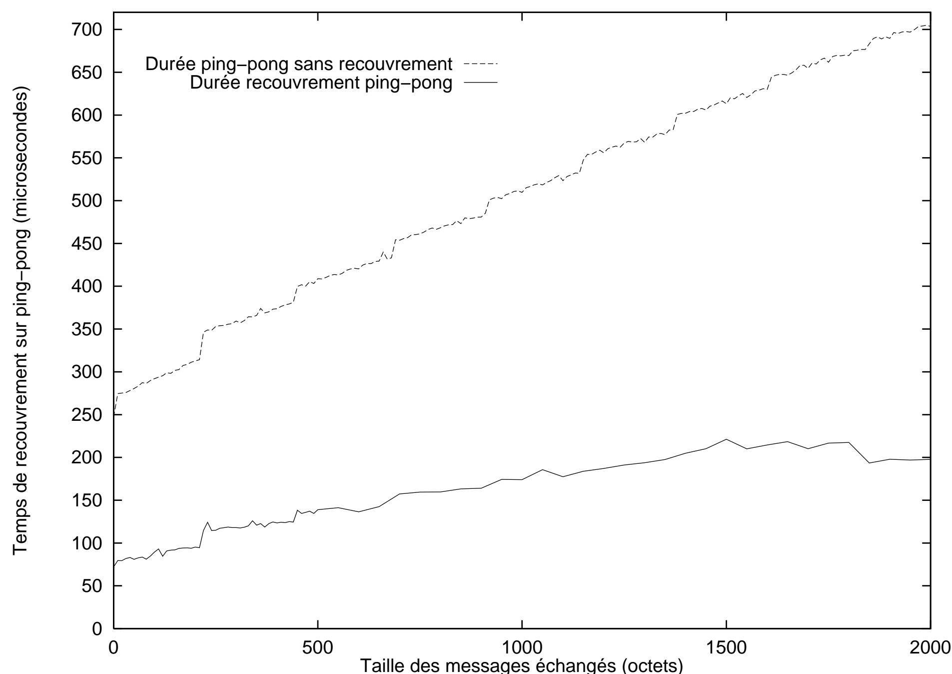

FIG. 7.14 – *Durée de recouvrement sur le nœud "ping" pendant un échange "ping-pong" complet (envoi et réception). Athapascan-0b sur AIX 3.2.*

Afin de mesurer le recouvrement effectué par Athapascan-0b entre des communications et des calculs, le programme de "ping-pong" de la section 7.2.3 est repris. Sur le nœud "ping", en plus du fil effectuant les communications s'exécute un autre fil effectuant des calculs. L'exécution de ces deux fils peut se faire de deux manières

différentes : soit le fil de communication s'exécute seul et quand il termine (quand l'échange "ping-pong" est terminé), s'exécute le fil de calcul, soit le fil de communication s'exécute en même temps que le fil de calcul. Le fil de calcul effectue de temps en temps des appels faisant avancer les communications, pour permettre au fil de communication de s'exécuter. Les temps d'exécution de ces deux versions du programme sont comparées pour chaque taille de message. La différence des temps d'exécution, ramenée à un échange "ping-pong", donne la durée des calculs pouvant être faits sur le nœud "ping" lors de chaque échange, sans retarder les communications. Il s'agit de calculs qui recouvrent les délais d'attente des communications. La figure 7.14 présente ces temps de recouvrement pour différentes tailles de messages.

On remarquera que diviser les valeurs de cette courbe par 2 pour obtenir le recouvrement ramené à une seule opération de communication n'a pas de sens, puisque les calculs recouvrent d'une part les communications mais également les attentes des communications originaires du nœud "pong". On sait que lors d'envois de messages des tailles de ceux tracés sur la figure 7.14, il n'y a pas blocage du fil émetteur, donc pas de recouvrement possible (ou plutôt un recouvrement "obligatoire" car quelle que soit la façon de programmer, la communication commencera à partir alors que les calculs continuent). Quand le fil "ping" se bloque ensuite en réception, il attend en fait que le message envoyé arrive au nœud "pong", que celui-ci réponde et que la réception soit prise en compte. C'est ce délai qui peut être recouvert par des calculs.

7.3.2 Appel et retour parallèles

On évalue ici la performance d'un appel parallèle avec création de fils distants. Il s'agit d'une structure utilisée fréquemment dans le calcul parallèle, dans laquelle un flot de contrôle se divise en créant n flots, qui effectuent des calculs puis renvoient leurs résultats au flot initial. Ici, les flots sont des fils d'exécution. On évalue la création par un fil initial de n fils d'exécution (sur n nœuds différents) et la remontée des résultats depuis ces fils vers le fil initial. La figure 7.15 illustre cette structure.

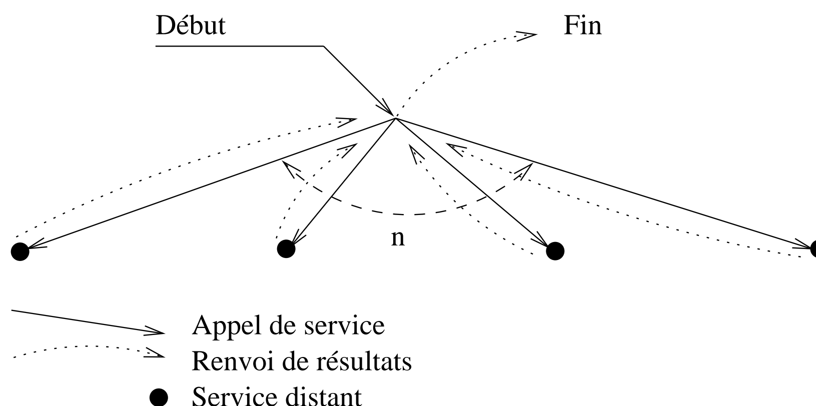

FIG. 7.15 – *Appel de n services par un fil et récupération des résultats.*

On mesure le temps total nécessaire pour créer des services distants, leur passer des paramètres et récupérer les résultats qu'ils renvoient. La durée des calculs que ces services peuvent effectuer n'est pas le sujet de la mesure, c'est pourquoi ces services ne calculent pas, et renvoient les résultats dès leur création. Des exécutions ont été mesurées pour un nombre de nœuds distants variant de 1 à 7 (en plus du nœud sur lequel s'exécute le fil initial), avec différentes tailles de paramètres et de résultats. Les figures 7.16 et 7.17 présentent les temps d'exécution, les paramètres passés aux services et les résultats renvoyés par eux ayant la même taille (en abscisse).

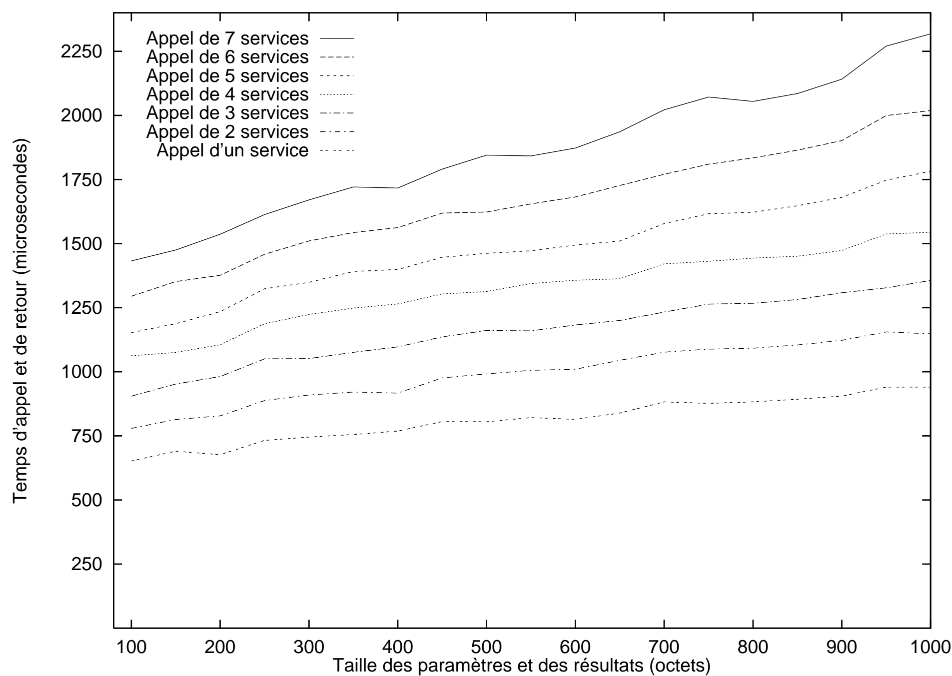

FIG. 7.16 – *Appel de n services et récupération des résultats, petits paramètres et résultats. Athapascan-0b sur AIX 3.2.*

Étudions en détail le temps d'exécution pour des petits messages, 100 octets par exemple. La courbe de la figure 7.8 page 101 indique que l'envoi d'un message de 100 octets par Athapascan-0b (dans un environnement non chargé), nécessite 150 microsecondes. La courbe de la figure 7.10 page 107 indique quant à elle que l'appel de service avec 100 octets de paramètres nécessite 480 microsecondes. La création d'un service et le renvoi de résultats par celui-ci, pour une taille de paramètres et de résultats de 100 octets, nécessite donc au minimum $480 + 150 = 630$ microsecondes. Il n'y a en effet aucun recouvrement possible, le renvoi des résultats étant une conséquence du début de l'exécution du service. Cette valeur (630 microsecondes) est une borne inférieure pour le temps d'exécution de cet exemple, quel que soit le nombre de services appelés. Si les appels de service se font de façon totalement synchrone – les résultats d'un appel sont attendus avant de passer au suivant – une borne inférieure du temps d'exécution pour n appels est $n \times 630$.

Ce n'est pas le cas du programme d'exemple utilisé, dans lequel les appels de service sont tous faits avant de commencer à attendre les résultats. Il y a recouvrement des différents appels. Toujours pour des messages de 100 octets, les courbes de la figure 7.16 permettent de donner une borne inférieure (car environnement non chargé) de $650 + (n - 1) \times 130$ microsecondes, soit 130 microsecondes par appel retour supplémentaire. En effet, l'appel du service et la récupération de ses résultats coûte pour un seul service 650 microsecondes et pour 7 services, 1430 microsecondes. $(1430 - 650)/6 = 130$. Il n'est pas possible à partir des courbes de déterminer qui, des appels de service et des réceptions des résultats, est responsable des 130 microsecondes de coût marginal. En d'autres mots, on ne sait pas si la cadence des appels de service ou celle des réceptions est le facteur limitant de la performance. La section suivante donne des éléments de réponse à cette interrogation.

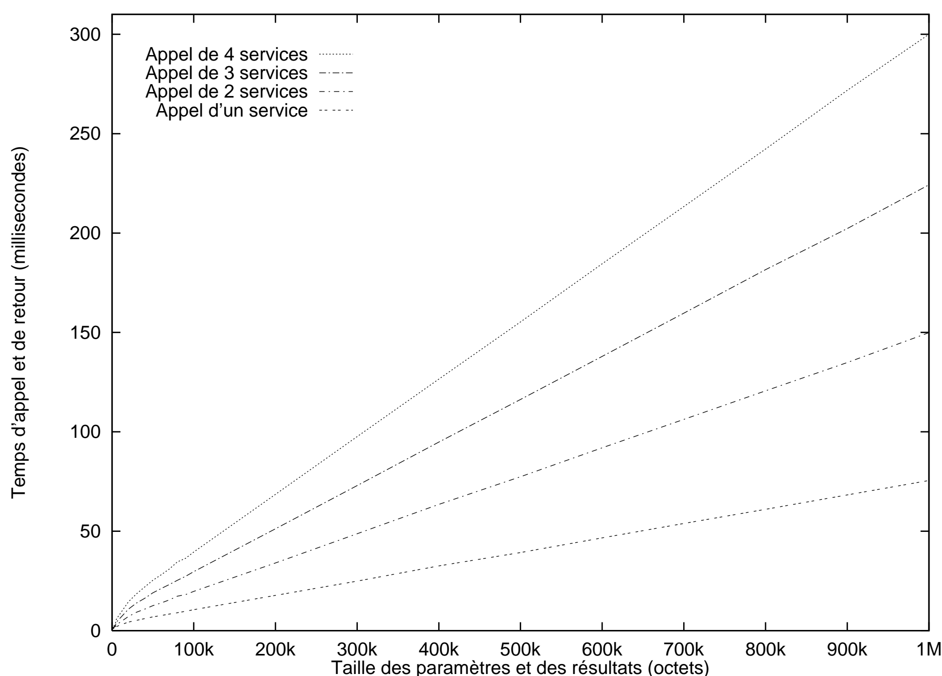

FIG. 7.17 – *Appel de n services et récupération des résultats. Athapascan-0b sur AIX 3.2.*

Dans les courbes de la figure 7.17, le coût dominant est celui des communications sur le réseau et les recouvrements des appels de service ou des réceptions sont négligeables (tout comme les coûts annexes associés à un appel de service). On retrouve donc les performances d'envoi de message d'Athapascan-0b sur AIX 3.2 telles que présentées en figure 7.7 page 101 (garder à l'esprit que la figure 7.17 présente la durée de l'appel et de la réception des résultats, qui équivaut donc à deux envois).

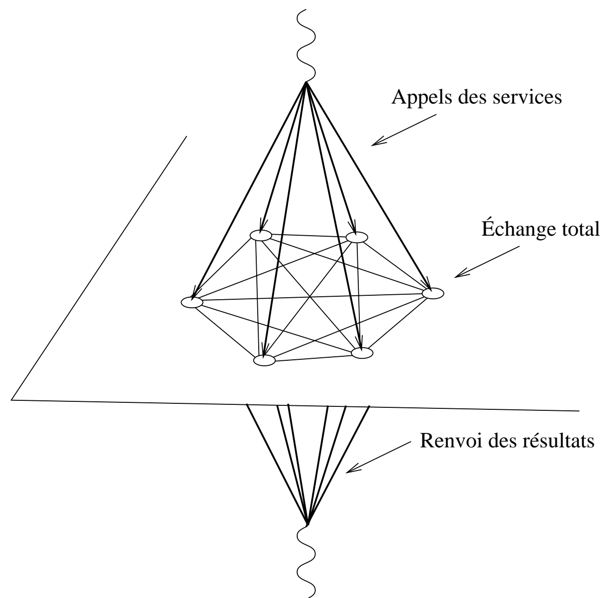

FIG. 7.18 – *Appels de services, échange total entre les services puis renvoi de résultats.*

7.3.3 Échange total

Si dans la section précédente un fil créait un ensemble de services qui ne communiquaient pas entre eux, ici on simule un échange total entre fils. La base de la simulation est similaire à celle de la section précédente : un fil principal crée des fils sur des processeurs par des appels de service. Cependant, ces fils effectuent entre eux un échange total avant de renvoyer un résultat au fil principal. L'échange total réalisé est "naïf", dans la mesure où chaque fil envoie un message à chaque autre fil, et reçoit un message de chaque autre fil (l'objectif ici n'est pas de proposer un échange total efficace mais d'étudier la performance d'un cas particulier). La figure 7.18 présente l'enchaînement des actions. Lors de la création des fils participant à l'échange total, le fil principal leur passe des paramètres de petite taille. Il n'a pas été jugé utile de faire varier cette valeur dans la simulation puisque cet aspect a déjà été traité dans la section précédente. Par contre, les tailles des messages de l'échange total ont varié.

La figure 7.19 présente les temps d'exécution de la simulation, quand le fil principal passe 100 octets de paramètres aux services créés, et que ceux-ci lui renvoient à leur terminaison également 100 octets de résultats. La taille des messages échangés lors de l'échange total varie et est portée en abscisse du graphique. L'implantation de l'échange total est directe : chaque fil envoie dans une boucle des messages à tous les autres fils puis fait une boucle de réception et se bloque ensuite sur l'attente de la fin des réceptions. Les communications sont asynchrones, ce qui permet à Athapascan-0b de les ordonner au mieux. On compare la performance de cette exécution avec celle de la section précédente, ne comprenant que les appels de service et le retour des résultats, ainsi qu'avec la performance de l'envoi de message (section 7.2.3 page 100).

Prenons l'exemple de deux service appelés qui échangent ensuite des messages de 100 octets. La durée totale de cette exécution est de 1000 microsecondes. Cette durée est à rapprocher d'une part de la durée d'installation et de remontée de paramètres de deux appels de service (760 microsecondes avec des paramètres et des résultats de 100 octets échangés avec le fil principal, voir figure 7.16 page 113) et d'autre part à la durée de communications entre deux fils de nœuds différents (150 microsecondes pour une taille de 100 octets, voir figure 7.8 page 101).

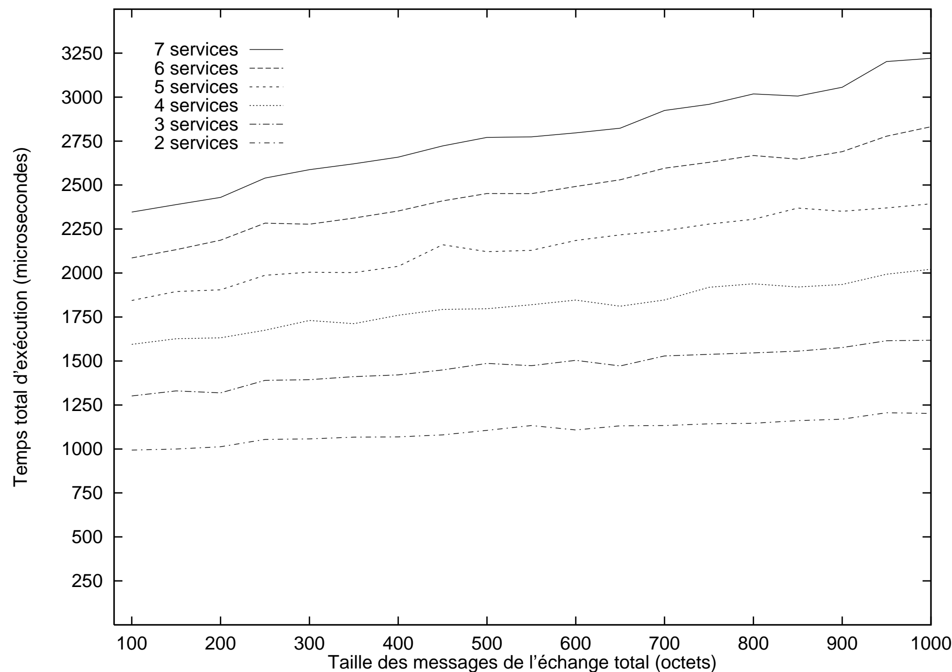

FIG. 7.19 – *Appels de services, échange total puis renvoi de résultats. Athapascan-0b sur AIX 3.2.*

Dans l'exemple à deux appels de service, quand on soustrait à 1000 microsecondes le coût d'installation des calculs et de remontée des résultats précédemment trouvé soit 760 microsecondes, il reste 240 microsecondes. Pendant ce temps se font les communications de l'échange total (dans ce cas précis, un envoi et une réception par chacun des deux fils), mais cette durée est trop longue pour n'être attribuable qu'aux communications. Une hypothèse est qu'il y a un retard à la remontée des résultats des services vers le fil initial dû à la synchronisation implicite effectuée par l'échange total. Dans le cas de la section précédente, où les services ne communiquaient pas, dès qu'un service était créé il renvoyait des résultats vers le fil initial et se terminait. Les résultats arrivaient sur le fil initial décalés dans le temps, suivant l'ordre séquentiel de création des services par le fil initial.

Dans le cas de l'échange total (toujours avec deux services), le premier fil créé est retardé par l'attente du second : le premier fil envoie un message à destination du second puis se bloque en attente de réception. Le second fil, à son arrivée, envoie un

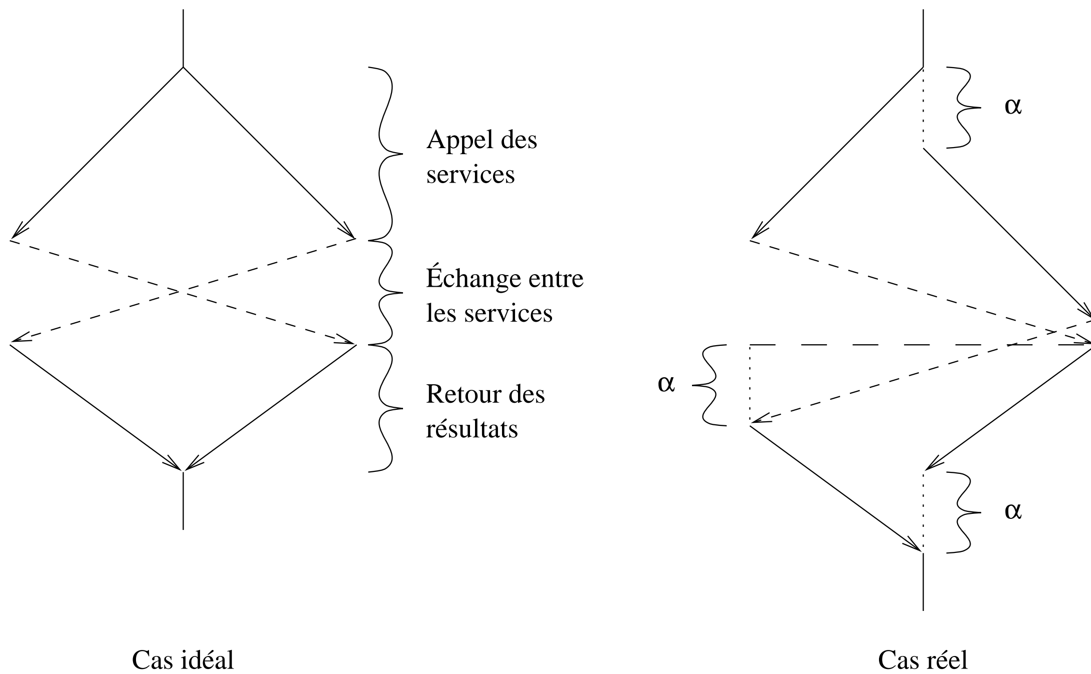

FIG. 7.20 – *Propagation du retard de création de service lors d'un échange total entre deux services.*

message à destination du premier puis effectue la réception qui ne se bloque pas (le message est déjà arrivé puisque le premier fil l'a envoyé en avance). Le deuxième fil complète donc son exécution avant le premier. On sait que globalement le surcoût par fil créé est de 130 microsecondes, payé à l'appel et au retour dans un rapport inconnu. Supposons le retard à l'appel de α microsecondes. Le deuxième fil est donc créé α microsecondes après le premier. Le retard induit sur l'échange total est alors également de α microsecondes, et se manifeste par un retard de la terminaison du premier fil créé (voir figure 7.20). Le retard de la création des services est propagé le long du calcul.

Cette justification du retard permet de calculer la valeur de α. En effet, l'échange total rajoute 240 microsecondes de temps d'exécution par rapport à la création des services et au retour des résultats. De ces 240 microsecondes, si 150 sont consommées par les communications de l'échange total, $\alpha = 90$ microsecondes. Ce résultat permet d'affiner l'analyse de la section précédente, puisque des 130 microsecondes que coûte chaque nouvel appel de service et retour des résultats, 90 seraient imputables à l'appel et $130 - 90 = 40$ au retour.

L'adjonction d'un nouveau nœud augmente le temps d'exécution de 270 microsecondes (2350 microsecondes avec 7 services appelés et 1000 avec 2. $(2350-1000)/5 = 270$). Ce temps comprend l'augmentation de la durée d'installation des calculs (130 microsecondes, déduite à partir de la figure 7.16) et le temps nécessaire à chaque fil de chaque nœud pour envoyer et recevoir un message supplémentaire lors de l'échange total. 140 microsecondes seraient alors nécessaires pour gérer ces communications sup-

plémentaires, ce qui implique qu'elles se recouvrent avec d'autres communications car la durée d'une communication isolée de 100 octets est de 150 microsecondes. Ce coût des communications tend à confirmer l'hypothèse précédente comme quoi le délai de 240 microsecondes n'est pas imputable aux seules communications.

7.4 Bilan

Dans ce chapitre la performance d'Athapascan-0b a été analysée. Tout d'abord comparée sur des primitives simples à celle des bibliothèques support utilisées, elle a été ensuite évaluée sur des construction plus complexes. Deux remarques peuvent être faites.

La première concerne les primitives d'Athapascan-0b ayant montré une efficacité moyenne comparativement aux couches support. L'analyse en détail de la performance de ces primitives a permis de comprendre où se perdait du temps lors de l'exécution. Cette vision de l'exécution d'Athapascan-0b n'était pas disponible lors de la réalisation ce qui explique que malgré les efforts faits alors, certaines parties sont perfectibles.

La deuxième remarque concerne la combinaison de fonctionnalités. Les exemples étudiés mettant en jeu plusieurs primitives ont montré que le comportement d'Athapascan-0b était “sain”, dans la mesure où non seulement il n'y a pas eu d'écroulement de performance pendant la montée en charge, mais des recouvrements se sont mis en place et la performance d'un ensemble d'actions s'est avérée supérieure à la performance des actions le composant prises une par une. Cette propriété permet d'envisager l'exécution d'applications complexes au dessus d'Athapascan-0b.

La programmation des exemples utilisés dans ce chapitre s'est faite facilement, confortant l'idée que les primitives offertes par Athapascan-0b permettent une écriture aisée de programmes parallèles.

Enfin, il devrait être possible de trouver des fonctions de coût donnant le temps total d'exécution en fonction du nombre de fils impliqués et de la taille des données échangées pour les expériences de ce chapitre. Ces fonctions seraient alors la généralisation des études faites à des tailles de messages et des nombres de fils d'exécution quelconques.

Chapitre 8

Conclusion et perspectives

Athapascan-0b a été développé dans le cadre du projet APACHE (algorithmique parallèle et partage de charge) et en constitue l'exécutif de base support des autres développements. Pour faciliter son utilisation, une importante documentation a été écrite, allant du manuel de l'utilisateur et de référence à la définition de l'interface de bas niveau (Akernel). Un site[1] a été créé sur la toile (*web*) où l'on peut consulter la documentation en ligne ou télécharger la dernière version du logiciel. Athapascan-0b est utilisé à travers le monde et nous avons notamment eu des contacts avec un utilisateur en Corée.

Plusieurs développements ont été faits au dessus ou autour d'Athapascan-0b :

– Athapascan-1 [16, 26, 27] est un langage qui implante des concepts de type *macro-dataflow*. Il permet la construction dynamique d'un graphe de dépendance de tâches, l'ordonnancement automatique de celles-ci et l'équilibrage de charge.

– Une application de dynamique moléculaire [3, 2, 4] permettant la simulation de problèmes à n-corps.

– Une sur-couche d'Athapascan-0b, appelée "Athapascan-0b Formats" a été développée [76]. Elle offre les concepts de flots de communications et d'accès de mémoire à distance.

– Une instrumentation de l'exécutif a été faite [91], permettant une prise de traces fines à des fins de visualisation post-mortem de l'exécution d'une application.

– Un environnement de visualisation exploitant les traces a été développé [21]. La figure 8.1 présente un exemple de visualisation de traces prises lors de l'exécution de l'application de dynamique moléculaire.

– Des travaux sur la réexécution déterministe [30] pourraient être adpatés à Athapascan-0b.

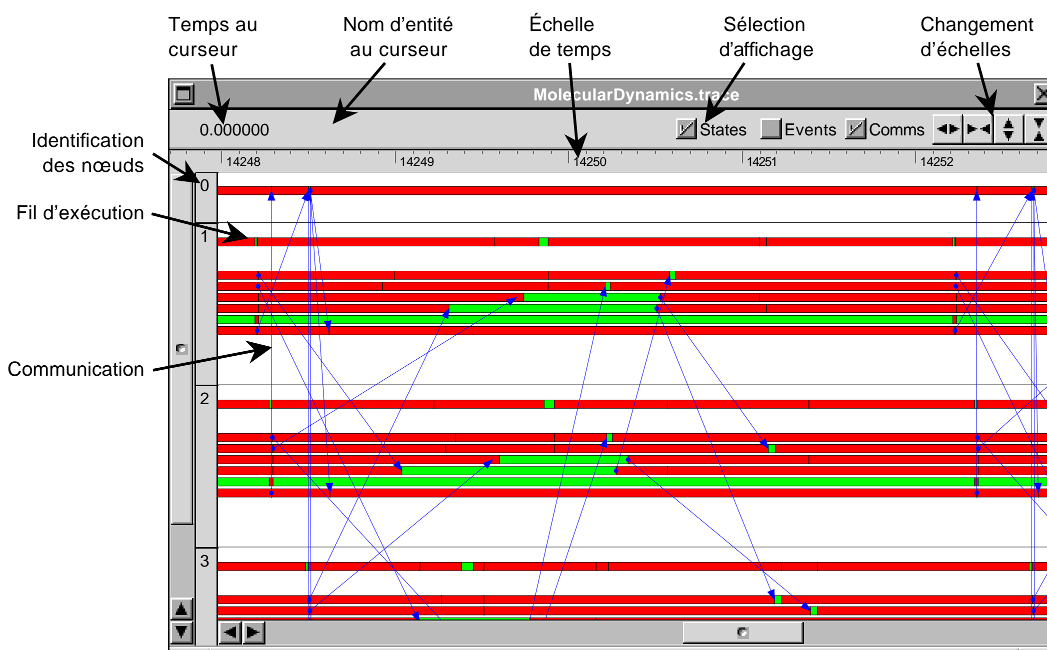

FIG. 8.1 – *Visualisation de traces prises lors de l'exécution de l'application de dynamique moléculaire.*

Les fonctionnalités offertes par Athapascan-0b se sont montrées suffisantes et d'une utilisation aisée pour les développements précédemment cités (couvrant un grand nombre de concepts de programmation parallèle). Les fonctionnalités actuelles telles que décrites dans ce document sont le résultat d'itérations avec les développeurs utilisant Athapascan-0b. Ces fonctionnalités sont simples et peuvent servir de support à des constructions plus complexes si besoin est. Elles ne cachent pas la structure de la machine (nœud local, nœud distant, communications, etc...) et affichent de ce fait clairement leur coût d'utilisation. Ceci permet au programmeur d'effectuer des choix tout en gardant la maîtrise de l'efficacité de son application.

L'argument de portabilité d'Athapascan-0b s'est révélé pleinement justifié. Les nombreux portages effectués n'ont pour la plupart posé aucun problème pour avoir une première version s'exécutant correctement. Des optimisation ont été faites sur certains portages et ont permis d'avoir de performances du même ordre que celles rapportées au chapitre 7 relativement aux bibliothèques utilisées. Athapascan-0b conçu à l'origine pour l'utilisation de processus légers de type *Pthreads* a été facilement porté au dessus d'autres noyaux.

L'efficacité d'Athapascan-0b peut être améliorée sur certains points, comme cela a été signalé dans les chapitres précédents. Ces améliorations sont faciles à mettre en œuvre. Un aspect qui par contre est difficile à améliorer est la réactivité des com-

1. `http://www-apache.imag.fr/ath0b/`

munications. L'utilisation de bibliothèques de communication et de multiprogrammation légère distinctes fait que l'interaction entre ces deux aspects n'est pas optimale. De meilleures performances pourraient être obtenues en intégrant à un niveau plus bas communications, multiprogrammation légère et le système d'exploitation. Dans des applications où la performance est un facteur critique, une telle intégration peut s'avérer intéressante. Il ne faut néanmoins pas perdre de vue qu'un tel développement nécessite des moyens importants et que pour une grande majorité d'applications, la performance d'exécutifs tels qu'Athapascan-0b est suffisante.

Il semble que les environnements intégrant communication et multiprogrammation légère sont appelés à un bel avenir, du fait d'une part de l'orientation actuelle vers des machines multi-processeurs à mémoire commune (SMP) et d'autre part des besoins croissants en puissance de calcul. Une interconnexion de telles machines par l'intermédiaire de réseaux hauts débits est un support idéal pour un exécutif tel qu'Athapascan-0b qui permet d'exploiter les différentes formes de parallélisme (voir section 5.1 page 61).

Il est tout à fait imaginable que la multiprogrammation et les communications seront intégrés à un bas niveau dans les systèmes futurs et seront présents sur les machines au même titre que le système d'exploitation. Il serait intéressant de conduire des recherches sur les problèmes soulevés par une telle intégration et sur son adaptation notamment aux nouveaux réseaux rapides.

Acronymes

ADI : Abstract Device Interface.

API : Application Programming Interface.

DCE : Distributed Computing Environment.

DTMS : Distributed Tasks Management System.

FIFO : First In, First Out.

GP : Global Pointer.

HPF : High Performance Fortran.

HPS : High Performance Switch.

IEEE : Institute of Electrical and Electronics Engineers.

LAM : Local Area Multicomputer.

MIMD : Multiple Instruction Multiple Data.

MPI : Message Passing Interface.

MPL : Message Passing Library.

OSF : Open Software Foundation.

PASC : Portable Application Standards Committee.

PM2 : Parallel Multithreaded Machine.

POSIX : Portable Operating System Interface for computer environments.

PVM : Parallel Virtual Machine.

RPC : Remote Procedure Call.

RSR : Remote Service Request.

SIMD : Single Instruction Multiple Data.

SMP : Symmetric Multi Processor.

SPMD : Single Program Multiple Data.

TCP : Transmission Control Protocol.

UDP : User Datagram Protocol.

Bibliographie

[1] 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 Principles*, 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 92.

[2] P.E. Bernard, B. Plateau, and D. Trystram. Using Threads for developing Parallel Applications: Molecular Dynamics as a case study. In Trobec, editor, *Parallel Numerics*, pages 3–16, Gozd Martuljek, Slovenia, September 1996.

[3] P.E. Bernard and D. Trystram. Algorithme Parallèle de Dynamique Moléculaire. Rapport APACHE 20, LMC-IMAG Grenoble France, June 1996.

[4] P.E. Bernard, D. Trystram, and Y. Chapron. Parallélisation d'un Algorithme de Dynamique Moléculaire. In *Actes RenPar8, Bordeaux*, pages 97–100, June 1996.

[5] M. Bever, K. Geihs, L. Heuser, M. Mühlhäuser, and A. Schill. Distributed systems, OSF DCE, and beyond. *Lecture Notes in Computer Science*, 731:1–20, 1993.

[6] Jacques Briat, Ilan Ginzburg, and Marcelo Pasin. Athapascan-0b reference manual. Technical report, Apache, LMC-IMAG, Grenoble, France, 1997.

[7] Jacques Briat, Ilan Ginzburg, and Marcelo Pasin. Athapascan-0b user manual. Technical report, Apache, LMC-IMAG, Grenoble, France, 1997.

[8] Jacques Briat, Ilan Ginzburg, Marcelo Pasin, and Brigitte Plateau. Athapascan runtime: efficiency for irregular problems. In *Proceedings of Euro-Par'97*, Aug 1997.

[9] R. A. A. Bruce, J. G. Mills, and A. G. Smith. CHIMP/MPI user guide. Technical Report EPCC-KTP-CHIMP-V2-USER 1.2, Edinburgh Parallel Computing Centre, 1994.

[10] G. Burns, R. Daoud, and J. Vaigl. LAM: an open cluster environment for MPI. In J.W. Ross, editor, *Proceedings Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[11] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley professional computing series, 1997.

[12] R. Butler and E. Lusk. Monitors, message, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.

[13] 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.

[14] Bryan M. Cantrill. Runtime performance analysis of the M-to-N scheduling model. Technical Report CS-96-19, Brown University, Department of Computer Science, May 1996.

[15] Ben Catanzaro. *Multiprocessor system architectures*. Prentice Hall, 1994.

[16] Gerson Cavalheiro and Mathias Doreille. ATHAPASCAN: a C++ library for parallel programming. In *Stratagem'96*, page 129, Sophia Antipolis, France, July 1996. INRIA.

[17] Michel Christaller. *Vers un support d'exécution portable pour applications parallèles irrégulières: Athapascan-0*. PhD thesis, Université Joseph Fourier, Grenoble, France, 1996.

[18] L. J. Clarke, R. A. Fletcher, M. Trewin, R. Alasdair, A. Bruce, A. G. Smith, and S. R. Chapple. Reuse, portability and parallel libraries. Technical Report TR9413, Edinburgh Parallel Computing Centre, University of Edinburgh, 1994.

[19] J.N. Colin. *DTMS: Un environnement pour la programmation distribuée à grain indéterminé*. PhD thesis, University of Mons-Hainaut, Belgium, 1995.

[20] Stephen Crane. The REX lightweight process library. Computer science technical report, Imperial College of Science and Technology, London, England, December 1993.

[21] Benhur de Oliveira Stein and Jacques Chassin de Kergommeaux. Environnement de visualisation de programmes parallèles basés sur les fils d'exécution. In *RenPar'9*, Lausanne, Suisse, mai 1997.

[22] Digital Equipment Corporation. *Guide to DECthreads*. Digital Equipment Corporation, Maynard, Massachusetts, March 1993.

[23] Thomas W. Jr. Doeppner. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Brown University, Department of Computer Science, Providence, RI 02912, June 1987.

[24] Jack Dongarra, G. A. Geist, Robert Manchek, and V. S. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in physics*, 7(2):166–174, March-April 1993.

[25] Jack Dongarra, Steve W. Otto, Marc Snir, and David Walker. An introduction to the MPI standard. Technical Report CS-95-274, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, January 1995.

[26] Mathias Doreille, François Galilée, and Jean-Louis Roch. Athapascan-1b: Présentation . Technical report, Projet APACHE, Grenoble, France, 1996.

[27] Mathias Doreille, François Galilée, and Jean-Louis Roch. Graphe de flot de données ATHAPASCAN et ordonnancement. In *RenPar'9*, Lausanne, Suisse, mai 1997.

[28] Susan J. Eggers, Joel Emer, Henry M. Levy, Jack L. Lo, Rebecca Stamm, and Deam M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. Technical Report UW-CSE-97-04-02, University of Washington, Department of Computer Science and Engineering, 1997.

[29] 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, Jun 1992.

[30] Alain Fagot. *Aide à la mise au point d'applications parallèles basées sur les processus légers*. PhD thesis, Université Joseph Fourier, Grenoble, France, 1997.

[31] T. Fahringer, M. Haines, and P. Mehrotra. On the utility of threads for data parallel programming. In ACM, editor, *Conference proceedings of the 9th International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995*, pages 51–59. ACM Press, New York, NY 10036, USA, 1995.

[32] Michael Feeley, Jeffrey Chase, and Edward Lazowska. User-level threads and interprocess communication. Technical Report UW-CSE-93-02-03, University of Washington, Department of Computer Science and Engineering, Feb 1993.

[33] Dror G. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, October 1994, Revised Version, February 1995.

[34] 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. IEEE catalog no. 95TB8075.

[35] Raphael A. Finkel. *An operating systems vade mecum*. Prentice Hall, Englewood Cliffs, NJ US, 1988.

[36] Jon Flower and Adam Kolawa. Express is not just a message passing system Current and future directions in Express. *Parallel Computing*, 20(4):597–614, April 1994.

[37] Message Passing Interface Forum. MPI: A Message Passing Interface. *Proceedings of the Supercomputing Conference*, pages 878–885, November 1993.

[38] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, June 95.

[39] Ian Foster, Carl Kesselman, and Steven Tuecke. Nexus: runtime support for task-parallel programming languages. Argonne National Laboratory, 1994.

[40] Ian Foster, Carl Kesselman, and Steven Tuecke. Portable mechanisms for multithreaded distributed computations. Argonne National Laboratory, 1994.

[41] 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.

[42] H. Franke, P. Hochschild, P. Pattnaik, and J.-P. Prost. MPI-F: an MPI prototype implementation on IBM SPI. In Jack J. Dongarra and Bernard Tourancheau, editors, *Environments and tools for parallel scientific computing: 2nd Workshop, May 1994, Townsend, TN*, pages 43–55, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[43] H. Franke, P. Hochschild, P. Pattnaik, J.-P. Prost, and M. Snir. MPI on IBM SP1/ SP2: current status and future directions. In IEEE, editor, *Proceedings of the 1994 Scalable Parallel Libraries Conference: October 12–14, 1994, Mississippi State University, Mississippi*, pages 39–48, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press.

[44] H. Franke, P. Hochschild, P. Pattnaik, and M. Snir. MPI-F: An efficient implementation of MPI on IBM-SP1. In Dharma P. Agrawal, K. C. (Kuo Chung)

Tai, and Jagdish Chandra, editors, *Proceedings of the 1994 International Conference on Parallel Processing, August 15–19, 1994. Vol 3: Algorithms and applications*, pages III–197–III–201, 2000 Corporate Blvd., Boca Raton, FL 33431, USA, 1994. CRC Publishers.

[45] H. Franke, C. E. Wu, M. Riviere, P. Pattnaik, and M. Snir. MPI programming environment for IBM SP1/SP2. In IEEE, editor, *Proceedings of the International Conference on Multimedia Computing and Systems, May 15–18, 1995, Washington, DC*, pages 127–135, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press. IEEE catalog no. 95TH8066.

[46] 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.

[47] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency, practice and experience*, 4(4):293–312, June 1992.

[48] G. Al Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8(2):137–150, June 1996.

[49] W. Gropp and E. Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Preprint MCS-P342-1193, Argonne National Laboratory, 1994.

[50] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[51] M. Haines and W. Böhm. An initial comparison of implicit and explicit programming styles for distributed memory multiprocessors. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 2: Software Technology*, pages 379–389, Los Alamitos, CA, USA, January 1995. IEEE Computer Society Press.

[52] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of chant: A talking threads package. In *Proceedings of Supercomputing '94*, 1994. Also as Tech report NASA CR-194903 ICASE Report No. 94-25, Institute for Computer Applications in Science and Engineering, NASA Langley Research.

[53] Matthew Haines, Piyush Mehrotra, and David Cronk. Chant: lightweight threads in a distributed memory environment. Technical report, Institute for Computer Applications in Science and Engineering, NASA Langley Research, June 1995.

[54] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: a case study. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 94–105, Ashville, NC, USA, Dec 1993. Published in ACM Operating Systems ReviewVol.27, No.5, Dec. 1993.

[55] Mark Heuser. An implementation of real-time thread synchronization. In *Proceedings of the Summer 1990 USENIX Technical Conference and Exhibition*, pages 97–106, Anaheim, CA, 1990. USENIX.

[56] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[57] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 18(2):95, February 1975. Corrigendum.

[58] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1984.

[59] Institute of Electrical and Electronic Engineers, Inc. *Threads Extension for Portable Operating Systems*. IEEE Standard 1003.4a, New York, N.Y., 1990.

[60] Institute of Electrical and Electronic Engineers, Inc. *Information Technology - Portable Operating Systems Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment 2: Threads Extension [C Language]*. IEEE Standard 1003.1c-1995, New York, N.Y., 1995.

[61] Michael B. Jones. Bringing the C libraries with us into a multi-threaded future. In *Proceedings of the Winter 1991 USENIX Technical Conference and Exhibition*, pages 81–91, Jan 1991.

[62] Steven Kleiman, Devang Shah, and Bart Smaalders. *Programming with threads*. Prentice Hall, 1995.

[63] Jeff Kramer, Jeff Magee, Morris Sloman, Naranker Dulay, S. C. Cheung, Stephen Crane, and Kevin Twindle. An introduction to distributed programming in REX. In *Proceedings of ESPRIT 91*, pages 207–222, Brussels, 1991.

[64] Doug Lea. *Concurrent Programming in Java*. Addison Wesley, 1997.

[65] Bill Lewis and Daniel Berg. *Threads primer*. Prentice Hall, 1996.

[66] Jay Littman. Applying threads. In *Proceedings of the Winter 1992 USENIX Technical Conference and Exhibition*, pages 209–221, San Francisco, CA, USA, Jan 1992. USENIX Assoc.

[67] D. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–42, 1993.

[68] Robert J. Manchek. Design and implementation of PVM version 3. M.s. thesis, University of Tennessee, Knoxville, Knoxville, TN 37996, USA, 1994.

[69] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 110–121, Pacific Grove, CA, October 1991. Published in ACM Operating Systems Review Vol.25, No.5, 1991.

[70] Oliver A. McBryan. An overview of message passing environments. *Parallel Computing*, 20(4):417–444, April 1994.

[71] Frank Mueller. Implementing POSIX threads under UNIX: Description of work in progress. In *Proceedings of the 2nd software engineering research forum*, pages 253–261, Melb., Florida, November 1992.

[72] Frank Mueller. A library implementation of POSIX threads under unix. In *Proceedings of the Winter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, USA, January 1993.

[73] Raymond Namyst. *PM^2: un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, France, 1997.

[74] Charles Northrup. *Programming with Unix threads*. Wiley, 1996.

[75] Parasoft Corporation. *Express Version 1.0: A Communication Environment for Parallel Computers*, 1988.

[76] Marcelo Pasin. Athapascan-0b Formats. publication à venir, 1997.

[77] P. Pierce. The NX/2 operating system. In *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[78] P. Pierce. The NX message passing interface. *Parallel Computing*, 20(4):463–480, April 1994.

[79] P. Pierce and G. Regnier. The Paragon[TM] implementation of the NX message passing interface. In P. Pierce and G. Regnier, editors, *Scalable high performance computing conference: — May 1994, Knoxville, TN*, Scalable High Performance Computing Conference, pages 184–190, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[80] Jon B. Postel. User Datagram Protocol. Request for Comments 768, DDN Network Information Center, SRI International, August 1980.

[81] Jon B. Postel. Transmission Control Protocol. Request for Comments 793, DDN Network Information Center, SRI International, September 1981.

[82] 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.

[83] R. Rashid, R. Baron, A. Forin, D Golub, M. Jones, D. Orr, and R. Sanzi. Mach: a foundadtion for open systems. In *Second Workshop on Workstation Operating Systems (WWOS-II)*, Pacific Grove, CA, 1989.

[84] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The design and evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April 1994.

[85] G. S. Sohi, S. E. Breach, and T. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[86] Dan Stein and Devang Shah. Implementing lightweight threads. In *Proceedings of the Summer 1992 USENIX Technical Conference and Exhibition*, pages 1–10, San Antonio, TX, 1992. USENIX.

[87] C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker. The SP2 high performance switch. *IBM Systems Journal*, 34(2):185–204, 1995.

[88] C. B. Stunkel, D. G. Shea, D. G. Grice, and P. H. Hochschild. The SP1 high-performance switch. In IEEE, editor, *Proceedings of the Scalable High-Performance Computing Conference, May 23–25, 1994, Knoxville, Tennessee*, pages 150–157, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[89] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, practice and experience*, 2(4):315–339, December 1990.

[90] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: thread pipelining with run-time data dependence checking and control speculation. In *1996 International Conference on Parallel Architectures and Compilation Techniques*, October 1996.

[91] Philippe Waille. Traçage d'Athapascan-0b. publication à venir, 1997.

[92] David W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, April 1994.

[93] M. Weiser, A. Demers, and C. Hauser. The portable common runtime approach to interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 114–122, December 1988.

Résumé

Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications

Athapascan-0b est un noyau exécutif pour machines parallèles supportant la multiprogrammation légère. Athapascan-0b permet un développement portable d'applications parallèles irrégulières et une exécution efficace de celles-ci sur un grand nombre de plates-formes.

Ce document commence par la présentation du cadre dans lequel s'inscrit Athapascan-0b, à savoir les communications, la multiprogrammation légère et l'intégration de ces deux fonctionnalités. Sont ensuite présentés les concepts structurant Athapascan-0b ainsi que son interface de programmation. La problématique d'une intégration de communications et de multiprogrammation légère est posée, la réalisation d'Athapascan-0b est décrite et plus précisément le choix d'implantation, à savoir un "mariage" de bibliothèques existantes de multiprogrammation légère et de communications.

Enfin, la performance d'Athapascan-0b est évaluée, comparée à la performance des bibliothèques au dessus desquelles il a été développé. L'exécution de quelques exemples est analysée afin de mieux comprendre les mécanismes en jeu.

Mots clés : calcul parallèle, communications, multiprogrammation légère, noyau exécutif parallèle, portabilité.

Abstract

Athapascan-0b: efficient and portable integration of communications and multithreading

Athapascan-0b is a multithreaded parallel programming runtime system. It is designed to support portable development and efficient execution of irregular applications on a large number of architectures.

This document starts by presenting the framework in which Athapascan-0b is inscribed, that is communications, multithreading and the integration of these two aspects. Then the concepts of Athapascan-0b are described as well as its application programming interface. The major difficulties in integrating communications and multithreading are mentioned, the implementation of Athapascan-0b is described and more precisely the implementation choice, consisting in the use of existing communication and multithreading libraries.

The performance of Athapascan-0b is then evaluated and compared to the performance of the underlying libraries. The execution of some toy programs is analyzed for a better understanding of the mechanisms involved.

Keywords: parallel computing, communications, multithreading, parallel runtime system, portability.