Vers un support d'exécution portable pour applications parallèles irrégulières: Athapascan-0

Michel Christaller

THÈSE

présentée par

Michel CHRISTALLER

pour obtenir le grade de DOCTEUR

de l'Université Joseph Fourier, Grenoble I

(*arrêté ministériel du 30 mars 1992*)

Spécialité : Informatique

Vers un support d'exécution portable pour applications parallèles irrégulières : Athapascan-0

Date de soutenance : 6 Novembre 1996

Composition du jury

Président :	Brigitte Plateau
Rapporteurs :	Jean-Marc Geib
	Traian Muntean
Examinateurs :	Jacques Voiron, *directeur de thèse*
	Jacques Briat, *co-directeur de thèse*

Thèse préparée au sein du
Laboratoire de Modélisation et de Calcul
(Institut d'Informatique et de Mathématiques Appliquées de Grenoble)

THÈSE

présentée par

Michel CHRISTALLER

pour obtenir le grade de DOCTEUR

de l'Université Joseph Fourier, Grenoble I

(*arrêté ministériel du 30 mars 1992*)

Spécialité : Informatique

Vers un support d'exécution portable pour applications parallèles irrégulières : Athapascan-0

Date de soutenance : 6 Novembre 1996

Composition du jury

Président :	Brigitte Plateau
Rapporteurs :	Jean-Marc Geib
	Traian Muntean
Examinateurs :	Jacques Voiron, *directeur de thèse*
	Jacques Briat, *co-directeur de thèse*

Thèse préparée au sein du
Laboratoire de Modélisation et de Calcul
(Institut d'Informatique et de Mathématiques Appliquées de Grenoble)

Ma compassion sincère
pour tous les thésards
qui n'en ont pas encore fini

Je tiens à remercier ici
les célèbres et les anonymes
qui individuellement ou collectivement
œuvrent pour doter la planète
d'une informatique libre de droits

Je remercie tout particulièrement :

- Madame Brigitte Plateau, responsable du projet APACHE, sans laquelle ce travail n'aurait pu exister,
- Messieurs Jean-Marc Geib et Traian Muntean, pour avoir bien voulu évaluer mon travail dans un temps limité, et pour leurs conseils avisés,
- Messieurs Jacques Voiron et Jacques Briat, pour m'avoir donné ma chance,
- et Eric, Martha, Michel, Ilan, ainsi que tous les autres apaches, pour l'aide qu'ils m'ont prodigué et pour l'intérêt qu'ils ont manifesté à l'égard de ce travail.

Table des matières

Table des figures

Liste des tableaux

Note :

un index (page 173) complète ce manuscrit. Les mots repris dans cet index sont définis dans le texte, et signalés par une fonte grasse.

Chapitre 1

Introduction

Dans ce chapitre, nous allons rapidement introduire la nécessité de maîtriser le parallélisme par l'étude des tendances des applications et des ordinateurs, puis nous présenterons ce qui constitue le corps de ce travail, à savoir l'élaboration d'un support d'exécution pour applications parallèles irrégulières et enfin nous replacerons cette thèse dans son contexte de réalisation et nous indiquerons sa motivation et ses objectifs.

1.1 Nécessité du parallélisme

Nous définissons ici une **machine parallèle** comme étant un ensemble d'unités de traitement qui coopèrent pour résoudre un problème. C'est une définition large qui englobe à la fois les super-calculateurs de plusieurs centaines de processeurs, les réseaux de stations de travail et aussi les machines symétriques multiprocesseurs. Dans ce contexte, le « **parallélisme** » est un champ d'études de l'informatique qui vise à permettre un fonctionnement optimum des machines parallèles. Les critères d'optimalité sont souvent l'efficacité d'une application parallèle sur une machine parallèle donnée - le pourcentage de chacun des processeurs de la machine qu'elle utilise réellement - et aussi l'extensibilité des problèmes qui peuvent être traités - le fait qu'une machine avec le double d'unités de traitement puisse résoudre des problèmes deux fois plus grands. Jusqu'à récemment le parallélisme était considéré comme un champ spécialisé de l'informatique, élaboré dans le cadre de besoins particuliers comme ceux des mathématiciens du calcul numérique. Nous allons montrer rapidement dans la suite pourquoi le parallélisme est important pour les années à venir.

1.1.1 Tendances des applications

On pourrait supposer que les ordinateurs finiront un jour par devenir suffisamment puissants pour que la demande de machines plus performantes disparaisse. Cependant, l'histoire montre que le développement d'une technologie satisfait la demande en cours, mais que de nouvelles demandes surgissent, entraînant un autre saut technologique. Traditionnellement, les super- calculateurs ont été utilisés pour des applications d'ingénierie et de calcul scientifique. Maintenant, les descendants des premiers super-calculateurs sont employés par une nouvelle demande qu'ils ont rendu possible, des applications commerciales comme les grandes bases de données, la réalité virtuelle, la vidéo à la demande, les environnements de travail collectif, ...

Même dans le domaine de l'ingénierie, la tendance est à remplacer les expérimentations par des simulations, pour diminuer les coûts et accroître la précision des résultats. La demande en calcul s'accroît typiquement comme la 4ème puissance de la précision du résultat. Les simulations demandent

donc des quantités de calcul, mais aussi des capacités de mémorisation et de manipulation de l'espace simulé toujours croissantes. La tendance des applications est donc d'utiliser toujours plus de puissance de calcul, d'espace de mémorisation et de capacités d'entrée-sortie.

1.1.2 Tendances des ordinateurs

1.1.2.1 Les microprocesseurs

L'évolution de l'électronique fait que les performances des ordinateurs les plus rapides augmentent exponentiellement depuis 1945, d'un facteur 10 tous les 5 ans, grâce à la diminution de taille des transistors. Cependant la technologie électronique actuelle est en train d'atteindre une limite imposée par la vitesse de la lumière. Pour contourner cette limite, les concepteurs tentent d'intégrer des opérations de plus en plus « grandes » (calcul sur 16 puis 32 et maintenant 64 bits en un cycle machine, par exemple). Toutefois, pour diminuer le temps requis pour une opération, il faut généralement accroître quadratiquement la surface du processeur [Ullman 1984]. Cette approche pose donc aussi des contraintes technologiques.

A l'inverse si l'on se contente de processeurs courants, pour la même surface de silicium on peut réaliser n^2 processeurs courants à la place d'un seul processeur, n fois plus rapide car effectuant des opérations n fois plus grandes dans le même temps de cycle. Ces n^2 processeurs courants sont idéalement capables d'effectuer n^2 opérations courantes en parallèle et sont donc (idéalement) n fois - plus rapides qu'un processeur unique de même surface.

Les concepteurs utilisent donc des techniques comme le *pipelining* (plusieurs unités de calcul effectuant des morceaux d'opérations en parallèle) et le *super-pipelining* (plusieurs unités de calcul effectuant des opérations entières en parallèle) pour profiter du parallélisme au niveau du microprocesseur (voir à ce propos [Goossens & Vu 1996]). Une autre solution consiste à interconnecter plusieurs processeurs complets - on obtient ainsi une machine parallèle.

1.1.2.2 Les réseaux d'interconnexion

Parallèlement à l'évolution des microprocesseurs (et en fait, résultant de cette évolution), les réseaux d'interconnexion offrent des débits de plus en plus grands, d'un niveau brut proche de celui du bus qui relie mémoire et processeur. Dans ces conditions il est naturel de concevoir des ordinateurs distribués, les processeurs étant reliés par un réseau d'interconnexion rapide. La surface de silicium, et donc la performance du super-calculateur, peut ainsi être accrue indépendamment de l'évolution de la technologie électronique, par simple ajout de processeurs.

En conclusion, les conditions sont réunies pour que le « parallélisme » devienne un champ important de l'informatique : saut quantitatif des besoins des applications à la fois en vitesse et en capacité de mémorisation, difficulté de construire des processeurs plus rapides, mais possibilité d'en agréger un grand nombre, interconnectés par un réseau rapide.

1.2 Le support d'exécution parallèle

Comment programmer un tel super-calculateur offrant plusieurs dizaines à plusieurs centaines de processeurs ? L'algorithme parallèle doit être décrit formellement dans un « langage » ou plus généralement à l'aide d'une interface de programmation. Le programme résultant sera exécuté sur la machine parallèle grâce à un support d'exécution - ou « *exécutif* » - particulier.

L'**exécutif parallèle** est la couche du « système d'exploitation » de la machine parallèle qui permet l'exécution d'un programme parallèle. Cet exécutif doit fournir des fonctionnalités pour :

- gérer les ressources physiques et virtuelles de la machine,
- gérer (créer, manipuler, détruire) des activités parallèles,
- gérer les communications et synchronisations entre ces activités,
- observer le comportement de l'application parallèle et éventuellement agir sur ce comportement.

L'exécutif parallèle peut être intégré au système d'exploitation de la machine parallèle : on parle alors de **système d'exploitation parallèle**. Cette approche a souvent été prise pour le développement de systèmes d'exploitation spécialisés pour des machines parallèles. A l'inverse, la machine parallèle peut être construite à partir de composants standards - y compris le système d'exploitation qui est alors classique et ne gère pas très bien les fonctionnalités nécessaires aux applications parallèles. Le support du parallélisme est alors à la charge d'une sur-couche de support d'exécution, distincte du logiciel de base proprement dit. Nous reviendrons beaucoup plus en détail sur les exécutifs parallèles au chapitre 2.

Notons ici qu'une application parallèle est semblable sous bien des points de vue à une application distribuée. Cependant un exécutif parallèle se distingue d'un support d'exécution distribuée - appelé parfois « système d'exploitation distribué » - par l'absence ou la moindre importance de fonctionnalités telles que la tolérance aux pannes, les protections et les droits d'accès, le nommage et la localisation des composants ou encore l'hétérogénéité. En contrepartie, l'exécutif parallèle se concentre sur la performance et l'extensibilité. L'un des points-clés concerne la transparence. Dans un système d'exploitation distribué, les différentes entités sont accédées de façon transparente ; l'utilisateur n'a pas besoin de connaître leur localisation, ni même leur existence. Dans un exécutif parallèle, la transparence n'existe généralement pas ; l'utilisateur doit connaître la localisation des différentes entités pour améliorer les performances. La frontière entre les deux mondes est cependant floue et difficile à tracer avec exactitude, puisqu'il peut arriver que la machine parallèle soit un réseau de stations, pour lequel certains problèmes des applications distribuées resurgissent et doivent être traités.

1.3 Motivation et objectifs de ce travail

La réalisation d'applications parallèles *régulières* - pour lesquelles le comportement est prévisible - est maintenant familière. Elle se base généralement sur un support d'exécution parallèle relativement proche de la machine physique qui, pour des machines à mémoire distribuée, fait intervenir les notions de *processus* et de *communication par échange de messages*[1] . L'ordonnancement des différentes tâches de l'application est alors établi en fonction d'une machine cible donnée et l'application parallèle est efficace sur cette machine.

Cependant la réalisation d'applications parallèles *irrégulières* est plus problématique. Nous définirons par le terme « **application parallèle irrégulière** » une application parallèle dont le comportement ne peut être prévu indépendamment des valeurs des entrées de l'application. C'est le cas par exemple de la plupart des applications qui font intervenir des structures de données creuses ou de tailles variables. En conséquence, il est difficile, voire impossible, d'établir un « bon » ordonnancement des différents sous-calculs avant l'exécution de chacun. Ce comportement particulier empêche l'emploi des techniques habituelles du parallélisme pour applications régulières sur machines à mémoire distribuée, à savoir la définition de communications structurées, synchronisées, comme celles présentées dans [Bala et al. 1992, Barnett et al. 1994].

1. Nous définirons ces termes au prochain chapitre

D'autre part l'évolution constante des machines parallèles conduit à un besoin de *portabilité de l'environnement de programmation*, et en particulier du support d'exécution. Elle conduit aussi à une plus grande *indépendance de l'application parallèle vis à vis de la machine cible*. Enfin l'observation et l'évaluation du comportement de l'application devient très important puisque ce comportement est changeant, en fonction soit des paramètres d'entrée, soit de la machine parallèle utilisée.

Il faut donc développer un environnement de programmation dédié aux applications parallèles *irrégulières*. Le projet APACHE[2] [Plateau & al. 1993, Briat et al. 1993] (acronyme d'Algorithmique Parallèle, progrAmmation et répartition de CHargE) mené au Laboratoire de Modélisation et Calcul de l'institut d'Informatique et de Mathématiques Appliquées de Grenoble s'est donné le but de *construire un environnement de développement d'applications parallèles permettant de résoudre des problèmes irréguliers* mais aussi des problèmes qui utilisent des structures de données non régulières ou variables au cours des calculs. Cet environnement de développement doit permettre d'une part d'*atteindre un bon compromis entre performance et portabilité* - puisque les machines parallèles évoluent très rapidement, il faut assurer la pérennité des applications en garantissant leur portabilité - et d'autre part de *construire des applications qui s'adaptent à la machine cible* - puisque le nombre de processeurs ne peut être fixé arbitrairement pour les générations de machines à venir, il est nécessaire que cette variabilité, ainsi que celle d'autres caractéristiques de la machine cible, soient prises en compte par l'application. En dernier lieu cet environnement doit *permettre l'observation et l'évaluation des programmes.*

Notre tâche au sein du projet APACHE a été de *concevoir et développer un support adéquat pour l'exécution d'applications parallèles irrégulières* sur des machines à mémoire distribuée. Ce support d'exécution n'emploie pas un modèle de communications collectives comme les supports pour applications régulières, mais suit un modèle de programmation assez original fondé sur le concept de *poly-algorithme* (2.4.3), qui nous semble adéquat pour assurer une bonne portabilité des applications parallèles, respectant l'efficacité. Le mécanisme de base de notre support d'exécution est un mécanisme d'appel de procédure à distance, *asynchrone*, car il permet une expression directe, mais de bas niveau, des poly-algorithmes et un recouvrement automatique des attentes de communication par de nouveaux calculs. L'ordonnancement de l'application irrégulière qui ne peut pas être défini statiquement est ainsi effectué « au mieux » par le support d'exécution, mais peut être contrôlé par l'application.

Ce support d'exécution doit être facilement portable sur différents systèmes de machines parallèles. Il s'appuie donc sur des « standards » : noyaux de communication par échange de messages d'une part, noyaux de multiprogrammation légère d'autre part. Le prototype qui a été réalisé est pleinement opérationnel et utilisé dans le cadre du projet APACHE comme support des activités des autres membres de l'équipe.

L'intérêt de ce travail est multiple et peut être résumé par les points suivants :

- identifier les problèmes à résoudre lors de la conception d'un tel support d'exécution ; en particulier nous avons mis à jour le problème de *l'ordonnancement- scrutation* dans le cas où le noyau de communication ne « remonte » pas d'interruptions,
- permettre la validation du modèle de programmation envisagé, à base de poly- algorithmes imbriqués, par la réalisation et l'étude d'applications réelles,
- montrer l'intérêt de l'utilisation de la multiprogrammation légère couplée aux noyaux de communication « usuels » comme PVM et MPI, et en particulier valider :
  - la facilité d'une telle implantation,

2. Le projet APACHE est projet INRIA depuis 1995 et a comme tutelles le Centre National de la Recherche Scientifique, l'Institut National Polytechnique de Grenoble, l'Institut National de la Recherche en Informatique et Automatique et l'Université Joseph Fourier.

  - le faible surcoût d'une telle implantation par rapport au noyau de communication natif,
  - le fait que l'introduction de la multiprogrammation légère ne contraint pas forcément à un mode de programmation compliqué, hors de portée du programmeur d'application,
- effectuer un état de l'art sur les supports d'exécution pour applications parallèles irrégulières, « postérieur » à notre travail, et donc élargir le débat et identifier les points de comparaison.

La présentation de notre travail suit le plan détaillé ici :

Le chapitre 2 effectue dans un premier temps un rapide survol des machines parallèles et introduit un modèle simplifié de machine et la notion de grain de parallélisme. Ensuite nous nous intéressons à l'expression d'applications parallèles. Nous identifions différents paradigmes de parallélisme et nous retenons l'importance du contrôle du grain quelque soit le paradigme employé. Nous identifions les supports exécutifs parallèles comme la base pour l'expression *explicite* du parallélisme. Nous indiquons différents modèles d'expression, à travers les notions d'interactions couramment utilisées entre les processus composants une application parallèle. Enfin nous introduisons le cadre de ce travail, c'est à dire les applications parallèles irrégulières et nous montrons la nécessité d'un support exécutif spécialisé pour ces applications. Nous décrivons les grandes lignes du support exécutif, nommé Athapascan, du projet APACHE.

Le chapitre 3 décrit en détail deux mécanismes utiles pour la réalisation d'un tel support exécutif : la multiprogrammation légère et les communications par échange de messages. Nous montrons des exemples précis, en particulier les noyaux employés pour la réalisation d'Athapascan.

Le chapitre 4 décrit la solution que nous avons retenue dans le cadre du projet APACHE : ses objectifs, son organisation, le modèle de programmation et l'interface résultante, l'implantation qui en a été menée et une analyse de l'efficacité du prototype obtenu. Nous concluons par l'expérience que nous a apporté ce prototype.

Le chapitre 5, par la présentation de divers exécutifs « concurrents » qui ont évolué en parallèle ou postérieurement au notre, nous permet de prendre du recul et d'identifier un certain nombre de caractéristiques communes et les différentes options de réalisation de supports exécutifs pour applications parallèles irrégulières.

Enfin, le chapitre 6 conclut cette thèse par une étude des perspectives liées à ce genre de supports exécutifs.

Chapitre 2

Quel support pour des applications parallèles irrégulières portables?

Dans ce chapitre nous allons examiner quelles sont les machines parallèles ; nous établirons un modèle simplifié de machine parallèle. Dans un second temps nous nous intéresserons à l'expression des applications parallèles. A travers l'étude de différents exécutifs parallèles, nous montrerons la nécessité d'un exécutif spécialisé pour le support d'applications parallèles irrégulières. Nous compléterons ce chapitre par une présentation du projet APACHE et de son support d'exécution de base, Athapascan-0.

2.1 Les machines parallèles

Il existe différents types de machines parallèles. Nous allons rapidement présenter les grandes tendances que sont les machines à mémoire partagée et les machines à mémoire distribuée synchrones ou asynchrones. Nous préciserons quelques-unes des caractéristiques principales de ces machines et nous définirons, comme support de notre discours ultérieur, un modèle simplifié de machine parallèle.

2.1.1 Machines parallèles à mémoire partagée

Dans une **machine parallèle à mémoire partagée** chaque processeur voit et partage un espace de programme et de données commun.

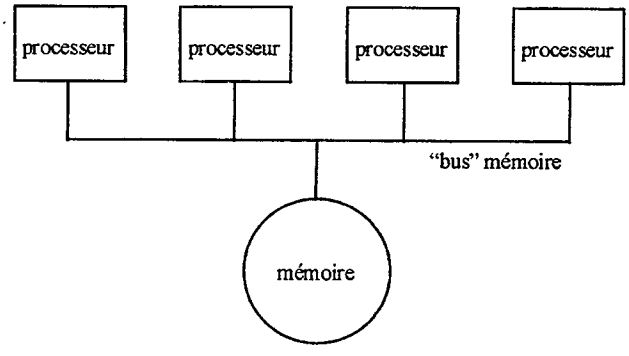

FIG. 2.1 - *Machine parallèle à mémoire partagée.*

Le principal avantage de cette architecture est que l'accès aux données est semblable à celui d'un programme séquentiel. Par exemple, le même élément de matrice peut être accédé par

Mat[ligne][colonne] depuis n'importe quel processeur. Un programme séquentiel peut donc être parallélisé sans modification du code, simplement en plaçant et synchronisant judicieusement les exécutions parallèles de blocs d'instructions indépendants.

Le principal inconvénient de ces machines réside dans l'accès à la mémoire qui est un goulot d'étranglement. Une solution répandue consiste à établir des caches locaux des zones mémoires utilisées. La plupart des accès se faisant alors dans le cache, le bus et la mémoire partagée sont déchargés. Le cache étant plus petit et plus rapide que la mémoire, il faut établir sa politique de gestion, visant à assurer les meilleurs temps d'accès. Chaque cache donne une vision locale de la mémoire partagée. Il faut donc s'assurer que cette vision est valide (problème de la **cohérence de cache**). Dans tous les cas, l'écueil du **faux partage** (*false sharing*), intervenant entre des données distinctes mais fortuitement gérées par les caches de façon couplées (dans la même *ligne de cache*), est possible. Deux processeurs peuvent donc écrouler le système des caches en accédant chacun ces données logiquement indépendantes.

Les machines à **mémoire physiquement partagée** sont habituellement bâties en agrégeant un certain nombre de processeurs identiques (avec leurs caches) autour d'un bus et d'une mémoire partagée. C'est pourquoi on les appelle « **machine SMP** » (*Symetric Multi Processing*), chaque processeur pouvant réaliser n'importe quel traitement. Les machines SMP sont maintenant bien répandues. L'efficacité des accès mémoire est relativement équilibrée, une donnée qui n'est pas trouvée dans le cache peut être accédée en 10 à 100 fois le temps d'accès d'une donnée dans le cache. Le principal problème, avec la technologie actuelle, est qu'il est difficile de dépasser une dizaine de processeurs ainsi couplés. Au delà, les bancs mémoire, le bus mémoire, ou bien les problèmes de cohérence de cache deviennent des goulots d'étranglement trop importants.

Une technique dérivée consiste à empiler hiérarchiquement les caches. La mémoire partagée est donc repoussée au sommet d'une arborescence de caches, voire supprimée (toutes les données résidant alors dans les différents caches[Lenoski et al. 1992, Hagersten et al. 1992]). Cela permet d'augmenter le nombre total de processeurs en réduisant les contentions ; en contrepartie les accès aux données sont nettement moins uniformes en fonction de leur localisation dans tel ou tel cache (on caractérise ces machines sous l'appellation **mémoire partagée NUMA** - *Non Uniform Memory Access*). Le principal problème est alors le respect de la localité des données et conduit donc à un résultat contradictoire avec l'avantage des machines à mémoire partagée : il faut tenir compte de la distribution des données sur les processeurs.

Certaines machines à mémoire distribuée (voir prochaine section) disposent d'une couche logicielle permettant de simuler une mémoire physiquement partagée : on parle alors de **mémoire virtuellement partagée** [Li & Hudak 1989, B.Carter et al. 1991, Z.Lahjmri & T.Priol 1992]. Les techniques employées font aussi appel à des caches locaux, des protocoles de maintient de la cohérence... Les problèmes avec cette approche ressemblent à ceux rencontrés dans l'implantation physique de la mémoire partagée : contention sur le médium limitant l'extensibilité de l'architecture, ou bien accroissement de l'importance de la localité, cassant en quelque sorte le modèle de programmation souhaité à l'origine.

Ces différents schémas de mémoire partagée peuvent s'appliquer avec un grand nombre de variantes [Kuntz 1993, Raina 1992, Sternström 1990]. Bien que solution performante pour quelques processeurs, nous ne considérerons plus ce type d'architectures dans la suite puisqu'elles sont difficilement extensibles ; au demeurant l'environnement dans lequel s'est déroulé ce travail ne comportait pas de machine importante de ce type.

2.1.2 Machines parallèles à mémoire distribuée

Dans une **machine parallèle à mémoire distribuée**, l'accès aux données n'est plus global : il y a distinction forte entre une donnée locale et une donnée distante que l'on ne peut pas accéder directement (on parle de **machine NORMA** - *No Remote Memory Access machine*). Il faut explicitement communiquer avec le processeur distant pour obtenir la donnée distante. Le programme séquentiel n'est donc plus utilisable tel quel. On doit rajouter, manuellement, les échanges de données entre les processeurs. En contrepartie, les problèmes liés aux caches ou aux contentions d'accès diminuent.

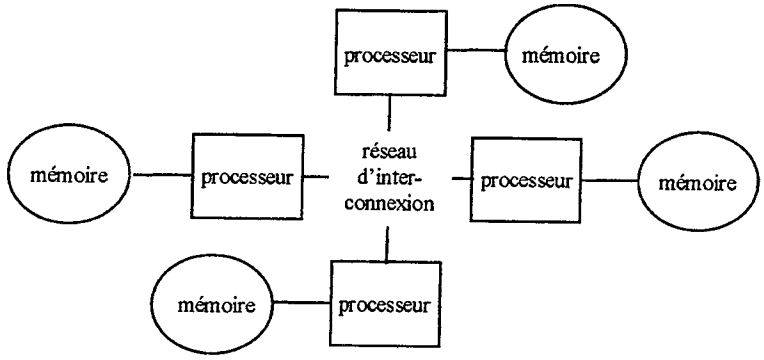

FIG. 2.2 - *Machine parallèle à mémoire distribuée.*

En dépit d'un coût de programmation plus élevé, ces machines permettent d'atteindre une plus grande efficacité à la fois parce que le programmeur sait exactement quelles données doivent être échangées et quand, ce qui entraîne un surcoût de gestion moins important, mais aussi parce qu'il est techniquement et économiquement possible d'agréger ainsi un plus grand nombre de processeurs. On peut distinguer deux grands types de machines parallèles à mémoire distribuée : les machines synchrones, dites SIMD (*Single Instruction Multiple Data*), et les machines asynchrones, dites MIMD (*Multiple Instruction Multiple Data*). Comme ces machines parallèles peuvent intégrer un grand nombre de processeurs, on les qualifie de **machine MPP** (*Massively Parallel Processing*).

2.1.2.1 Machines SIMD

Dans une **machine SIMD**, tous les processeurs sont cadencés par une horloge unique et exécutent la même instruction en même temps, sur leurs données locales qui sont souvent très petites (quelques bits). Typiquement, ce genre de machine est utilisée pour des calculs spécialisés comme le traitement d'images [Mohammadi et al. 1995], pour lequel ce fonctionnement est approprié. Les processeurs étants très simples, il est possible d'en intégrer un grand nombre en général sur une topologie de grille ou de tore.

Ces machines sont trop spécialisées pour servir dans le cadre du calcul parallèle en général. Nous n'y reviendrons pas dans la suite.

2.1.2.2 Machines MIMD

Une **machine MIMD** est plus généraliste ; elle a des processeurs plus puissants, des mémoires plus importantes et chaque processeur effectue ses propres calculs locaux indépendamment des autres. Les processeurs interagissent en transmettant des paquets d'information par l'intermédiaire d'un **réseau d'interconnexion** qui peut prendre plusieurs formes. Nous distinguerons deux générations différentes de ces machines : celles qui présentes une grande spécificité et celles qui sont assemblées à partir de composants standards.

Machines parallèles « d'ancienne génération » Dans l'ancienne génération, les machines parallèles sont composées de noeuds spécifiques, souvent à base de processeurs Transputers [Hinton & Pinder 1993]. Le système d'exploitation de ces machines est spécifique. Il permet la gestion des ressources de la machine, le partage entre plusieurs utilisateurs, l'interaction avec une « console » qui sert de point d'entrée dans la machine parallèle. Il offre aussi des mécanismes efficaces pour la gestion dynamique d'activités de calcul et la réalisation des communications. Par contre, ce système d'exploitation n'offre généralement pas d'émulation Unix, ni réellement de système de fichiers et manque d'outils de développement...

Dans ces machines, les caractéristiques du réseau d'interconnexion sont fortement visibles. Par exemple, une application tirera souvent partie d'une topologie de réseau particulière. Certaines de ces machines sont construites autour d'un réseau de proche en proche, dans lequel chaque processeur peut communiquer directement avec ses voisins immédiats mais plus difficilement avec les processeurs plus éloignés. Les communications entre voisins sont donc performantes. Pour assurer la communication au delà des voisins immédiats, il est nécessaire d'établir un mécanisme de **routage** des informations, effectué par chaque processeur, comme le *store and forward*. Le routage fut l'objet de nombreuses recherches [Bui 1994, Mugwaneza 1993, Upfal 1989].

Machines parallèles « de nouvelle génération » Le problème des machines d'ancienne génération est principalement économique : leur grande spécificité en fait des machines chères, dont l'investissement en programmes peut difficilement être conservé lorsqu'on passe d'une machine à l'autre.

En conséquence, la nouvelle génération de machines parallèles est construite autour de composants standards. Les noeuds sont des cartes, voire des châssis complets, de stations de travail. Les processeurs sont standards, généralement le haut de gamme du marché micro- informatique. Le système de ces machines est à la base classique : c'est un Unix complet sur chaque noeud, doublé d'outils d'administration globale de la machine parallèle. L'environnement de développement de ces machines est celui que l'on trouve habituellement sur une station de travail. Des outils spéciaux pour la programmation parallèle sont aussi fournis : soit des bibliothèques portables, garantissant une conformité à un « standard », soit des bibliothèques propriétaires, permettant une utilisation plus efficace des dispositifs matériels.

Deux classes peuvent être distinguées pour ces machines. La première est celle des machines parallèles intégrées, possédant un réseau d'interconnexion dont la géométrie est contrôlée et relativement statique, des noeuds souvent identiques. Ces machines sont en générales enfermées dans une « armoire ». Le réseau d'interconnexion de ces machines est souvent bâti à l'aide de commutateurs. Un **commutateur** (*switch*) [Tubtiang 1993, Konstandinidou & Upfal 1991] possède en général n entrées et n sorties et est capable de commuter simultanément les entrées vers les sorties s'il n'y a pas de collision (ie. deux paquets en entrée devant sortir par la même sortie). En cas de collision, le commutateur procède soit à un contrôle de flux, soit dispose d'une petite mémoire tampon pour cacher quelques collisions [Stunkel et al. 1994]. Les commutateurs peuvent être enchaînés ce qui leur donne une extensibilité quasi illimitée, au prix cependant d'une latence plus importante. Le débit de la communication est important, s'établissant à n fois le débit d'un lien point à point si les collisions sont peu fréquentes. Dans cette architecture, le routage est bien souvent figé par le constructeur, de même que la couche basse du protocole de communication (perte, déséquencement, contrôle de flux). Contrairement aux réseaux de proche en proche, la distance entre les processeurs n'intervient pas du fait du routage *wormhole* et du contrôle de la géométrie physique de la machine : le coût de communication est constant quelques soient les processeurs qui interagissent. Cette caractéristique, ainsi que le fait que les commutateurs sont devenus très rapides, explique le débit accru de ces nouveaux réseaux.

La seconde classe de machines parallèles de nouvelle génération est celle des réseaux de stations. (**réseau NOW**, *Network Of Workstations*). Ce sont les « machines parallèles du pauvre ». En effet,

c'est un équipement qui est relativement bon marché et très disponible dans un laboratoire ordinaire. En contrepartie l'hétérogénéité devient importante, en termes de :

- puissance de calcul des processeurs (et charge),
- format de données acceptées par les processeurs,
- adressage des données (structure mémoire différente),
- fonctionnalités (affichage graphique, serveurs de fichier ou d'application, ...).

Le réseau, dit « local », **réseau LAN** (*Local Area Network*) car il interconnecte des machines proches, est soit à bus (Ethernet fin), soit en anneau (*Token Ring*, *Fiber Distributed Data Interface*), soit en étoile ou en arbre (Ethernet 10baseT)... Bien que les propriétés de ces réseaux diffèrent, il sont tous « lents » relativement à ceux des machines parallèles intégrées et n'interconnectent efficacement que quelques dizaines de processeurs. Il est possible d'interconnecter différents réseaux locaux grâce à des ponts ou des routeurs, ce qui offre une extensibilité certaine à ce type de machine parallèle, au prix du routage entre les différents sous-réseaux. Le réseau obtenu est alors appelé **réseau WAN** (*Wide Area Network*) car il peut interconnecter des réseaux très distants, par satellite par exemple.

Une évolution récente consiste à utiliser la technologie des commutateurs, aussi bien en local (Ethernet commuté, réseau Myrinet) qu'à moyenne ou longue distance (Asynchronous Transfer Mode, ATM). La distinction entre machine parallèle intégrée et réseau de stations s'atténue donc de plus en plus. La différence essentielle reste la répartition géographique des nœuds, et donc le contrôle possible sur les caractéristiques du réseau d'interconnexion.

2.1.2.3 Les machines utilisées

Trois types de machines parallèles (de « nouvelle génération ») ont été employées dans le cadre de cette thèse ; nous décrivons les caractéristiques des nœuds, le type de réseau physique et les « couches de protocoles » utilisées :

- Des réseaux de stations - Sun sous SunOs 4 ou Solaris, compatibles PC sous Linux - avec réseau Ethernet fin. Utilisation de PVM domaine public sur IP sur Ethernet. C'est le réseau local habituel d'un laboratoire.
- La ferme de 16 Alphas du laboratoire LIFL de l'USTL. Processeurs DecChip 21064, 133Mhz, 512Ko de cache, 64Mo de mémoire, 1Go de disque par nœud. Chaque nœud tourne sous une version complète de DEC OSF/1. La ferme possède un réseau Ethernet et un réseau GigaSwitch (crossbar de fibres optiques, 200Mo/s par liaison, [Souza et al. 1994]). Utilisation de PVM domaine public sur IP sur Ethernet ou sur IP sur le GigaSwitch.
- L'IBM SP1.5 à 32 processeurs du LMC-IMAG. Processeurs Power1 à 60Mhz, 64Mo de mémoire, 1Go de disque, deux adaptateurs réseau Ethernet 10Mb/s et un adaptateur HighPerformanceSwitch TB2 par nœud. L'IBM 9076 SP1 est le premier ordinateur parallèle d'IBM largement commercialisé. Chaque nœud exécute une version complète d'Aix 3.2.5. Le *HighPerformanceSwitch* dérive du projet Vulcain du IBM Yorktown Research Center. Il est capable de soutenir 40Mo/s de transfert bidirectionnel avec une latence (physique) de 125ns. Utilisation de PVM domaine public sur IP sur Ethernet et sur IP sur le HPS. Utilisation de PVMe sur CSS-CI (*Communication SubSystem Client Interface*) sur le HPS.

2.1.3 Un modèle de machine parallèle à mémoire distribuée

Dans la suite nous considérerons le routage ainsi qu'une partie de la couche basse de communication comme figés et opaques. En conséquence nous supposons que le réseau d'interconnexion est

totalement maillé. Nous préciserons les propriétés particulières du réseau d'interconnexion et du protocole de communication, quand ce sera nécessaire. En général, nous considérerons la communication fiable (aucun message n'est perdu ou altéré) et sans déséquencement ni doublage (les messages émis par un processeur à destination d'un autre sont reçus par ce dernier dans leur ordre d'émission) et le contrôle de flux effectué de bout en bout (l'émetteur ne peut pas émettre tant que le récepteur ne veut pas recevoir).

Chaque nœud de la **machine MMP** disposera d'une mémoire privée. Cette mémoire pourra éventuellement être partagée entre plusieurs processeurs de calcul et plusieurs processeurs de communication, présents dans le nœud. A un moment donné, chaque processeur de calcul effectue au plus une **séquence d'instructions** de calcul, et chaque processeur de communication effectue au plus une **séquence d'instructions** de communication. Les processeurs (de calcul et de communication) d'un nœud peuvent se synchroniser par exemple à l'aide de la mémoire qu'ils partagent dans le nœud. Les processeurs de communication peuvent exister réellement ou bien plus généralement représenter une couche basse de communication, accédant par exemple à plusieurs liens de communication concurremment. De même les multiples processeurs de calcul peuvent être réels, ou bien plus généralement représenter le résultat de la multiprogrammation d'un unique processeur comme nous le verrons dans la suite. La machine parallèle peut être représentée schématiquement comme suit :

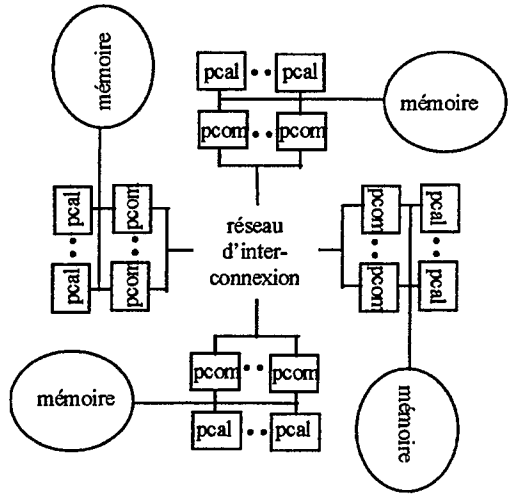

FIG. 2.3 - *Machine parallèle virtuelle « MMP » (Multi-Multiprocessors).*

Sur chaque nœud, plusieurs processeurs de calcul et de communication se partagent une mémoire commune

Dans notre schéma, chaque nœud de la machine est présenté comme identique. Cependant, il est possible que les nœuds diffèrent : par leur capacité mémoire, par le nombre de processeurs de calcul et le nombre de processeurs de communication qu'il est possible de faire fonctionner concurremment, par la puissance ou la charge de chacun, par le contrôle d'équipements spécialisés... De même le réseau d'interconnexion peut ne pas être d'un bloc uniforme mais être l'agrégation de divers réseaux ayant des propriétés différentes. Nous préciserons le moment venu l'importance de l'hétérogénéité de la machine.

2.1.4 Caractéristiques des machines parallèles à mémoire distribuée

Les machines parallèles à mémoire distribuée procèdent habituellement à l'échange de paquets d'information à travers leur réseau d'interconnexion. Le coût de cet échange peut généralement être approché par une règle linéaire :

durée d'envoi des informations = latence + taille des informations / bande passante

Cette règle est une approximation grossière de la réalité car elle néglige le phénomène de contenion ; cependant, pour la suite de notre discours, nous n'aurons pas besoin d'une modélisation plus récise. Notons que cette règle a été vérifiée expérimentalement sur les machines qui ont été utilisées voir 4.7.4). Elle peut être exprimée dans la métrique d'une opération élémentaire de calcul.

La **latence de communication** est la durée (ou le nombre d'opérations de calcul correspondant) our démarrer une communication ; la **bande passante de communication** définit le débit maximum 'une liaison bi-points ; on peut l'exprimer comme le nombre d'opérations de calcul correspondant la durée d'envoi (en régime stable) d'un paquet d'informations. Dans tous les cas l'échange d'inormations à travers le réseau d'interconnexion est plus coûteux que l'interaction directe de deux rocesseurs sur un même nœud (de 10 à plus de 100000 fois plus coûteux). En conséquence, l'utiliation du réseau de communication doit être contrôlée précisément.

Une caractéristique importante du réseau d'interconnexion est la taille des messages qui réalisent ne fraction donnée du débit maximal. Si l'on envoie des messages très petits, le débit obtenu du éseau d'interconnexion sera très faible. En conséquence, il est nécessaire de fixer une efficacité miimale et d'essayer d'envoyer toujours des messages qui réalisent cette efficacité minimale. La taille 'un message qui réalise une fraction x du débit maximal est donnée par la formule :

taille_min(x) = x/(1-x) * latence * bande passante

Par exemple pour disposer de la moitié du débit maximal, il faut employer des messages de taille lus grande que latence * bande passante. Nous appellerons cette taille la **taille de demi-débit**. Pour tiliser 80% du débit maximal, il faut des messages 4 fois plus grands...

Une autre propriété importante de notre machine parallèle est la **latence de calcul** pour démarrer n calcul parallèle sur un processeur, elle aussi pouvant être exprimée en nombre d'opérations de alcul correspondant. Enfin la dernière propriété est la **bande passante de calcul** ; elle est généraement exprimée en Mips ou Mflops, selon que l'on considère les instructions entières ou flottantes. xprimée dans la métrique d'une opération de calcul, elle est bien évidemment constante et égale à n, c'est pourquoi nous ne la considérerons plus.

Les trois paramètres, latence de communication, bande passante de communication et latence de alcul parallèle, influent sur ce que l'on peut appeler le grain de parallélisme souhaitable. En effet, i les séquences de calcul ou de communication sont trop courtes, les temps de latence nécessaires our les démarrer sont comparativement trop importants. Le **grain de parallélisme**, que l'on définira omme la longueur des séquences, est alors trop fin. A l'inverse si l'on regroupe trop les séquences et u'elles deviennent très longues, les latences sont négligeables, mais le nombre de séquences qu'il est ossible d'exécuter à un instant donné dans la machine parallèle se réduit beaucoup. En conséquence ertains processeurs de calcul ou certains processeurs de communication seront inutilisés. Le grain st alors trop gros et le parallélisme est restreint.

.2 L'expression d'applications parallèles

Nous avons dans la section précédente rappelé les caractéristiques des machines parallèles et préenté un modèle de machine, MMP. Nous avons identifié le problème de la granularité de parallélisme our une utilisation efficace de cette machine. Nous allons maintenant nous intéresser aux différentes açons d'exprimer une application parallèle, et en particulier son grain de parallélisme.

Comme nous l'avons précisé durant la présentation du modèle MMP, les processeurs d'un nœud e synchronisent par l'intermédiaire de la mémoire locale au nœud. Entre deux nœuds, les processeurs e synchronisent par des communications. Nous considérerons qu'après chaque synchronisation, une

nouvelle séquence de calcul est démarrée. Chaque séquence est donc « indivisible » ; les synchronisations marquant les frontières des séquences. Chaque séquence étant démarrée à la suite d'une synchronisation qui terminait d'autres séquences, on pourra établir un graphe, dit **graphe de dépendance**, entre les séquences. Une **application parallèle** s'exécutant sur notre machine sera donc déterminée par un ensemble de séquences à exécuter sur les différents nœuds et un graphe de dépendance entre ces séquences.

Une façon de concevoir une application parallèle est de l'exprimer avec un **parallélisme maximal**, c'est à dire en identifiant le plus de séquences possible de façon à mettre à jour tout le parallélisme latent. Cette expression n'est généralement pas suffisamment efficace. Il faut rassembler les séquences et les placer sur les processeurs de telle sorte que le grain de parallélisme soit adéquat, ni trop fin, ni trop gros. Deux objectifs contradictoires doivent être pris en compte lors de l'assemblage et du placement ; il faut :

- minimiser les communications (l'accès aux données doit être le plus local possible : on parle de la **localité d'accès aux données**),
- maximiser la concurrence (garder le plus de tâches actives en même temps : c'est l'**équilibrage de charge**)

La réalisation de ces deux objectifs est un problème difficile et crucial au même titre que celui du contrôle du grain. Nous ne décrirons pas dans cette thèse comment la localité des données ou l'équilibrage de charge peuvent être gérés. De nombreux ouvrages y font référence et nous invitons le lecteur à les consulter [Foster 1995, Carriero & Gelernter 1990]. Dans le cadre de cette thèse nous supposons que l'établissement d'une bonne localité et d'un bon équilibrage de charge est toujours possible indépendamment du grain lui-même. En conséquence dans la suite nous ne nous intéresserons qu'au contrôle du grain, qui doit être dynamique si l'on veut garantir une exécution efficace sur toute une gamme de machines parallèles. Le point important que nous retiendrons est qu'une application parallèle est exprimée sous forme de séquences de calculs et de communications et que le « bon » grain dépend des caractéristiques de la machine parallèle. Il n'existe pas un grain idéal, mais plutôt une plage en dehors de laquelle soit le surcoût de génération du parallélisme devient trop important, soit la machine parallèle n'est pas exploitée suffisamment.

2.2.1 Paradigmes de parallélisme, granularité et portabilité

Il est nécessaire d'exprimer les différentes séquences qui composent une application parallèle. Quatre grandes classes de paradigmes de parallélisme peuvent être distinguées :

- le **parallélisme de données** consiste à effectuer le même traitement en parallèle sur une collection de données. Chaque donnée est placée sur un processeur déterminé et le traitement est généralement effectué à cet endroit. Bien souvent ce paradigme de parallélisme se résume en une itération d'une phase de calculs sur les données locales et d'une phase de communications pour mettre à jour les données distantes. Une redistribution des données peut être effectuée périodiquement pour améliorer la localité et l'équilibrage.
- le **parallélisme de contrôle** (fonctionnel) consiste à découper un traitement en plusieurs traitements plus petits qui sont exécutés en parallèle. La découpe peut être « récursive », jusqu'à obtenir un traitement élémentaire qui est réalisé séquentiellement. Cette expression est adaptée lorsque le traitement à effectuer dépend fortement de la donnée fournie en paramètre. Dans ce cas c'est le traitement que l'on place sur un processeur ; les données sont soit accédées à distance, soit déplacées en tant que paramètres du traitement. Ce schéma permet de rendre le programme modulaire et donc de faciliter sa maintenance par la définition d'interfaces entre les différents modules le composant, interfaces indépendantes de leur réalisation.

- le **parallélisme d'acteurs** se produit lorsque l'application peut être décrite en terme d'entités qui interagissent. Les acteurs sont placés sur les processeurs ; les données sont soit internes à un acteur et elles caractérisent alors son comportement ; soit échangées entre les acteurs et elles définissent alors leurs interactions. Généralement cela correspond à une modélisation directe de la réalité du travail à effectuer. Ce schéma de parallélisme permet d'encapsuler dans des acteurs différents des comportements différents. Il se distingue du parallélisme de contrôle par le fait qu'il n'y a pas de décomposition d'un acteur en sous-acteurs parallèles : c'est la multiplicité des instances d'acteurs qui génère le parallélisme.
- enfin, le **parallélisme pipeline** consiste à appliquer une suite de traitements à une suite de données. Chaque processeur effectue un traitement particulier sur une donnée puis la transmet à son voisin. En bout de chaîne les données, sur lesquelles ont été appliqués tous les traitements, sortent une à une. Ce parallélisme est surtout utile lors de la réalisation de circuits matériels car il permet de profiter d'une exécution concurrente tout en optimisant chaque étage de calcul.

Quelque soit le paradigme de parallélisme employé, le problème du contrôle de la granularité se pose. Dans le parallélisme de données la granularité est déterminée par la finesse de la découpe des données. Dans le parallélisme de contrôle, la découpe récursive peut être arrêtée lorsque la séquence de calcul est devenue suffisamment petite. Dans le parallélisme d'acteurs, la granularité n'est contrôlable que si les acteurs sont des instances de grain choisi à partir de modèles d'acteurs génériques. Dans le parallélisme pipeline, la granularité est déterminée par le recouvrement calcul / communication optimal.

Ce choix de la granularité peut être fait de façon statique, en fonction des caractéristiques d'une machine parallèle cible particulière. Cependant, l'application parallèle n'aura pas alors forcément un grain adéquat pour une exécution efficace sur une autre machine parallèle. Pour supporter efficacement la portabilité d'une application sur différentes machines parallèles, il est nécessaire de pouvoir contrôler son grain de parallélisme dynamiquement en fonction de la machine cible, et ce quelque soit le paradigme de parallélisme employé.

2.2.2 La mise en évidence et les outils d'expression du parallélisme

On peut distinguer deux grandes classes d'expression d'applications parallèles selon que le programmeur doit explicitement décrire le parallélisme de son application ou non.

2.2.2.1 Parallélisme explicite

Le **parallélisme explicite** demande la formulation explicite de composants parallèles pour résoudre un problème. Les concepts de base du parallélisme explicite sont les concepts de processus et d'interaction entre processus. Nous reviendrons sur ces concepts, mais deux modèles peuvent être grossièrement distingués : un modèle à mémoire partagée, dans lequel tous les processus interagissent indirectement par la consultation ou la modification d'une mémoire qu'ils partagent ; et un modèle à échange de messages, dans lequel les processus interagissent par des communications directes de processus à processus.

Le modèle à échange de messages est très proche du modèle de machine physique que nous avons établi et est donc relativement simple à mettre en œuvre. De nombreux systèmes de programmation sont basés sur cette expression du parallélisme ; nous en présenterons certains dans la suite.

Le modèle à mémoire partagée nécessite la réalisation d'une mémoire logiquement partagée entre les processeurs et rencontre les mêmes problèmes d'extensibilité et / ou de cohérence de caches que les machines parallèles à mémoire physiquement partagée. En contrepartie la programmation est plus

simple puisque tous les processus ont accès aux mêmes données. Un certain nombre de systèmes de programmation proposent cette expression du parallélisme ([Carriero & Gelernter 1989, Cabrera-Dantart et al. 1994]), mais avec moins de succès actuellement en terme d'efficacité que ceux de l'autre forme.

Dans le parallélisme explicite, les processus concurrents qui définissent les différentes séquences de calcul parallèles et les interactions entre processus qui définissent les communications sont souvent établies d'une façon figée dans le programme. En conséquence le grain de parallélisme de l'application est figé lui aussi. Pour garantir la portabilité, il est nécessaire que le grain de parallélisme, et donc que *l'expression même de l'application*, soient dynamiques en fonction de la machine cible.

2.2.2.2 Parallélisme implicite

Le parallélisme existe intrinsèquement dans la plupart des formulations mathématiques d'un problème. Par exemple, l'algèbre booléenne autorise l'évaluation concurrente des deux opérandes d'un ET ou d'un OU. Plus généralement, tous les arguments d'une fonction peuvent être calculés en parallèle. De même, la définition de beaucoup d'objets mathématiques composés comme les vecteurs ou les matrices conduit à un parallélisme intrinsèque. Par exemple une somme de vecteurs peut être calculée en parallèle puisque chaque composant résulte d'une séquence de calculs indépendants. Le parallélisme sera dit **parallélisme implicite** s'il se base sur les propriétés intrinsèquement parallèles des objets et des opérateurs qu'il manipule.

En conséquence, une façon naturelle d'exprimer le parallélisme est tout simplement d'écrire un algorithme dans un formalisme « mathématique ». Le parallélisme sera implicitement établi au travers des opérateurs employés. Le parallélisme implicite est la base de systèmes de programmation dans différents domaines :

- le calcul formel,
- la programmation logique (typiquement, PROLOG),
- la programmation fonctionnelle (LISP et ses dérivés),
- le calcul numérique (les FORTRAN hautes performances en sont un exemple type).

Quelque soit le domaine et les opérateurs utilisés, une étape de traduction vers un langage à parallélisme explicite est *toujours* effectuée. Cette traduction peut être faite préalablement à l'exécution par un **compilateur- paralléliseur**, ou durant l'exécution par un support d'exécution adapté. Dans les deux cas l'application est traduite dans un langage à parallélisme explicite dans lequel les séquences concurrentes sont d'une granularité déterminée et sont placées sur les processeurs en essayant d'optimiser la localité d'accès aux données et l'équilibrage de la charge. Le contrôle du grain doit être établi d'une façon ou d'une autre. Cela peut être fait à l'aide d'un support d'exécution adaptatif et / ou d'annotations du programme pour le compilateur-paralléliseur, visant à agréger les séquences de calculs d'une façon convenable pour la machine cible.

Examinons maintenant les différentes classes d'outils qui permettent l'expression d'une application parallèle.

2.2.2.3 Outils d'expression du parallélisme et supports exécutifs parallèles

Les outils qui servent à exprimer le parallélisme sont de trois ordres :

- les langages explicitement parallèles,
- les langages séquentiels étendus d'opérateurs pour l'expression du parallélisme et
- les compilateurs paralléliseurs.

L'utilisateur d'un langage explicitement parallèle doit apprendre et utiliser les constructions particulières de ce langage, en général simples et adéquates, pour exprimer son application. Le créateur du langage, en regard de la souplesse d'expression qu'il peut offrir, devra fournir des outils pour transformer ce langage symbolique en un code exécutable. L'investissement effectué par le créateur dans la fourniture des outils et par l'utilisateur dans l'apprentissage du langage, pourra être compensé par la facilité d'expression de l'application (car bien souvent le langage influe sur le raisonnement). Le langage explicitement parallèle permet en particulier la manipulation automatique de types de données variés lors des interactions entre les composantes du programme parallèle et la vérification statique de ces types lors de la compilation.

Une approche plus pragmatique et moins lourde à réaliser est, plutôt que d'offrir un langage explicitement parallèle complet, d'offrir une bibliothèque permettant d'exprimer le parallélisme à l'intérieur d'un langage conventionnel en général séquentiel. La réalisation d'un traducteur code symbolique vers code exécutable est donc supprimée ainsi que l'apprentissage d'un langage complet, mais d'un autre coté la structure du langage séquentiel hôte peut se révéler contraignante. La vérification statique du type des données devient difficilement possible ; une approche orientée objets peut aider sur ce point.

Enfin un compilateur-paralléliseur prend en entrée une expression implicite du parallélisme et, à travers des règles de traduction, convertit cette expression en un parallélisme explicite. Ce genre d'outil est intéressant car il permet à l'utilisateur de continuer à penser d'une façon « séquentielle » ou de transformer un programme séquentiel préexistant en un programme parallèle. Si cette approche est particulièrement efficace dans le cas de traitements simples et réguliers, elle est moins adaptée aux applications irrégulières car généralement la complexité et l'aspect dynamique des interactions entre les composantes parallèles font qu'il est très difficile de réaliser une bonne parallélisation automatique, sauf dans des domaines particuliers.

Dans ce contexte, nous souhaitons mettre en avant les outils de **support d'exécution** (*runtime system*) qui sont réellement la base pour l'expression d'applications parallèles. En effet, aussi bien un langage explicitement parallèle qu'une extension parallèle d'un langage séquentiel, qu'un outil de génération automatique de code parallèle, produisent des séquences d'instructions qui seront en fait exécutées à l'aide d'une couche de support d'exécution. Le support exécutif parallèle regroupe donc l'ensemble des moyens permettant de réaliser l'exécution de l'application parallèle, indépendamment de l'expression qu'en fait l'utilisateur. C'est en quelque sorte une machine virtuelle. Pour prendre une analogie, l'exécutif correspond aux appels systèmes d'Unix, alors que la couche réellement visible de l'utilisateur correspond à l'interface de programmation d'Unix. Cette interface de programmation peut avoir différentes variantes comme la Spécification 1170 [Tristram 1995] ou bien BSD [Leffler et al. 1989]. L'exécutif à la base ne dépend pas obligatoirement de l'expression de l'application. Dans le cadre de cette analogie, nous voyons aussi que l'exécutif n'est généralement pas directement manipulé par l'utilisateur. Cependant il présente lui aussi une interface qui peut servir pour une programmation directe. En ce qui concerne le parallélisme, bien souvent l'exécutif n'est pas aussi clairement et précisément défini que les appels systèmes d'Unix ; il est caché derrière le langage parallèle, la bibliothèque de parallélisation ou le compilateur-paralléliseur. Chaque système de programmation parallèle possède généralement un support exécutif particulier qui est plus ou moins mis en évidence.

Les fonctionnalités de l'exécutif concernent principalement :

- la gestion des ressources physiques (choisir un processeur, un lien de communication...), et des ressources virtuelles (tâches, canaux de communication...),
- la gestion des activités concurrentes (les créer et les détruire, les identifier et les localiser...),
- la réalisation de leurs interactions (communications et synchronisations...),
- et l'observation et le contrôle de l'exécution (typiquement : réguler la charge).

La réalisation de ces actions dépend de la machine cible. Il est nécessaire de définir un support d'exécution adéquat pour une classe de machines, qui garantisse la portabilité des applications sur cette classe. La fonction première de l'exécutif est donc de permettre l'exécution d'applications sur toute machine parallèle qui fait partie de la classe cible.

Les fonctionnalités que doit offrir un exécutif dépendent de l'usage qui en sera fait, et en particulier des concepts manipulés lors de l'expression de l'application. Par exemple, si l'on décide d'exprimer l'application comme un ensemble de processus coopérants par échange de messages, alors l'exécutif doit offrir des fonctionnalités plus ou moins évoluées pour échanger des messages. Nous allons donc dans les deux sections suivantes présenter certains des concepts utilisés pour exprimer explicitement le parallélisme d'une application, ainsi que quelques exemples typiques, dans le but d'identifier les composants fondamentaux des exécutifs parallèles.

2.2.3 Modèles d'expression explicite du parallélisme

L'expression d'une application parallèle repose principalement sur l'expression de la *concurrence* entre activités, ou plus exactement sur l'expression du *parallélisme* de ces activités, c'est à dire de leur concurrence éventuelle et de la multiplicité des sites d'exécution possibles. Le concept qui nous permet d'exprimer des activités en parallèle indépendamment d'une technique particulière de réalisation est le concept général de *processus*. Il est excessivement rare qu'une application parallèle soit composée de processus indépendants. Après avoir présenté la notion de processus nous présenterons donc différentes notions *d'interaction* que l'on peut imaginer entre les processus. Ces notions d'interactions sont toutes fondées, pour une machines parallèle à mémoire distribuée, sur un mécanisme de base, d'échange d'informations. Nous appelons *canal de communication* l'équivalent des processus pour les communications, et nous présentons cette notion à la suite des processus.

2.2.3.1 Les processus

Un **processus** peut être défini comme un flot d'exécution séquentiel d'instructions. Un processus est donc composé :

- d'un ensemble d'instructions à exécuter,
- et d'un contexte de travail permettant de maintenir une « mémoire » reliant chaque instruction.

Un processeur peut n'exécuter qu'un seul processus ; plus généralement les processeurs sont multiprogrammés c'est à dire qu'une combinaison de mécanismes matériels et logiciels permet l'exécution pseudo-parallèle de plusieurs processus sur un unique processeur. Le contexte de travail est alors découpé en deux parties : un contexte *privé* nécessaire à l'exécution séquentielle des instructions d'un processus donné ; et un contexte *partagé* qui permet de réaliser l'interaction entre processus. Avec un système multiprogrammé il est courant que plusieurs utilisateurs utilisent un même processeur ; chaque utilisateur ayant ses propres processus. Il se pose alors un problème de protection du contexte privé des processus et de partage du contexte partagé. Cependant pour notre étude nous considérons que la machine est mono-utilisateur et nous n'entrerons pas dans les problèmes de protection d'accès.

Les processus peuvent être définis statiquement dans l'application parallèle et placés sur les processeurs lors du chargement de l'application. Alternativement, ils peuvent être définis dynamiquement lors du chargement ; par exemple si l'on souhaite qu'il y ait toujours exactement un processus par processeur. Mais généralement les processus sont démarrés dynamiquement, à la demande d'un autre processus qui peut être soit local - le processus fils réside sur le même processeur que son père - soit distant - la création est alors réalisée à travers le réseau d'interconnexion. Les mécanismes permettant de réaliser une création locale ou une création distante sont habituellement différents ; l'exécutif

qui doit réaliser cette opération dépend donc de l'expression choisie pour le modèle de création de processus.

Le contexte privé d'un processus est généralement conséquent ; il contient la pile des appels de fonctions en cours, les registres du processeur et des éventuels coprocesseurs utilisés, mais aussi un espace mémoire qui lui sert pour stocker les paramètres et les résultats de ses calculs. Pour différentes raisons que nous examinerons plus en détail par la suite, la manipulation de ce contexte privé est généralement très coûteuse lors de la création ou de la commutation d'un processus. C'est pourquoi ces processus sont appelés *lourds*. Une technique présentée au chapitre suivant permet l'utilisation de processus beaucoup plus *légers*, par la mise en commun entre plusieurs processus de la partie du contexte privé la plus lourde à manipuler. La création et la commutation de processus légers sont ainsi beaucoup plus rapides, ce qui autorise une exécution très dynamique de l'application parallèle. L'exécutif peut utiliser uniquement des processus lourds ou bien prendre en compte différentes implantations du concept plus récent de processus légers.

Lorsqu'un processeur est multiprogrammé, l'ordre d'exécution des processus sur le processeur est important. Habituellement l'exécutif qui réalise la gestion des processus possède un auto-ordonnanceur qui lui permet de réaliser le multiplexage des processus sur le processeur. Nous présentons un peu l'auto-ordonnancement de processus légers dans le chapitre suivant ; nous n'en dirons donc pas plus ici mais nous retiendrons que l'une des composantes d'un exécutif qui supporte la multiprogrammation est un auto-ordonnanceur, et qu'un point important est la façon dont cet auto-ordonnanceur peut être contrôlé par l'application.

2.2.3.2 Les canaux de communication

Dans notre description d'un modèle de machine MMP, nous parlions de séquences de calcul et de séquences de communication. Nous avons formalisé le concept de séquence de calcul sous le nom de processus. Nous allons maintenant formaliser le concept de séquence de communication.

Nous définirons un canal de communication d'une façon générale comme une suite de communications. Un canal de communication peut être établi entre deux ou plusieurs processus par une opération particulière que nous appellerons *connexion*, puis détruit par une *déconnexion*. A l'extrême la connexion peut être statiquement établie entre les processus ; il n'est alors pas possible de la couper ou d'en établir de nouvelles. A l'inverse, la connexion peut être dynamiquement établie lors de chaque communication et détruite tout de suite après.

Nous supposerons les canaux de communication fiables et, dans une certaine mesure, ordonnés. Différentes notions d'ordonnancement des communications peuvent être citées. La notion la plus restrictive est qu'il existe un ordre unique des communications sur le canal, et que tous les processus voient les communications dans cet ordre. Cette notion permet d'établir la cohérence lors d'interactions pluralistes, mais est relativement coûteuse à réaliser. Une notion plus relâchée est la cohérence vis à vis d'un processus : tout processus qui reçoit des communications d'un processus donné les reçoit dans leur ordre d'émission. Cependant, des communications provenant de plusieurs processus différents sont reçues dans n'importe que ordre. Enfin l'ordonnancement des communications peut être dirigé ; par exemple on peut vouloir volontairement s'occuper de certaines communications avant les autres. Cela peut être réalisé de deux façons : soit l'on filtre les messages à l'intérieur d'un canal, soit l'on utilise plusieurs canaux *indépendants*. Deux techniques de filtrage sont possibles. Le filtrage de bout en bout, qui consiste à retarder l'émission, revient à définir un canal indépendant par filtre. Le filtrage local, qui consiste à retarder seulement la réception, ne définit pas des canaux réellement indépendants : il est possible que les données en attente finissent par engorger le site récepteur et bloquent les communications qui ne sont pas filtrées.

Lors de l'échange d'informations entre deux processus, les données sont habituellement sorties du contexte de travail du processus émetteur, manipulées par la couche qui réalise le protocole de communication, transmises au dispositif matériel de communication, transportées sur le réseau d'interconnexion, reçues par le dispositif matériel de communication du site récepteur, manipulées par le protocole de communication une nouvelle fois, et finalement stockées dans le contexte de travail du processus récepteur. Tout ce parcours des données nécessite un certain nombre de copies, à la fois sur le site émetteur, sur le site récepteur et dans le réseau d'interconnexion. Ce nombre de copies peut être réduit par différentes techniques que nous n'aborderons quasiment pas ici. Une façon particulière d'éviter certaines copies est d'émettre directement les données depuis le contexte de travail, et de les recevoir directement dans le contexte de destination. Cela évite l'utilisation d'un tamponnage temporaire dans la couche du protocole de communication, mais nécessite une présentation particulière des données au protocole de communication. L'important est de retenir que la communication peut être plus ou moins lourde selon la réalisation et l'expression des communications.

Si l'on compare les canaux de communication aux processus, nous voyons une similitude dans les caractéristiques fondamentales :

- la création peut être prédéfinie, statique ou bien dynamique,
- la réalisation peut être lourde ou légère,
- l'ordonnancement et son contrôle peuvent être réalisés de différentes façons.

Dernier point important, le multiplexage des séquences de calcul et des séquences de communication doit être réalisé d'une façon ou d'une autre pour faire progresser de façon indépendante les communications entre entités communicantes indépendantes. Nous n'en parlerons pas plus ici, mais nous y reviendrons vers la fin de ce document.

2.2.3.3 Les principales notions d'interaction

La première interaction qui existe entre processus est la création dynamique de nouveaux processus et le passage d'informations entre le processus père et le processus fils, soit lors de la naissance du processus fils (flux du père vers le fils), soit lors de la mort du fils (flux du fils vers le père). Nous parlerons d'une **interaction dynamique** si certains correspondants sont créés durant l'interaction, d'une **interaction statique** sinon.

Comme nous l'avons dit, l'interaction entre deux processus quelconques peut se faire principalement selon deux modes : par échange de messages ou à travers une mémoire partagée. De façon plus détaillée nous parlerons d'une **interaction directe** si les deux processus interagissant se connaissent, d'une **interaction semi-directe** si un seul connaît l'autre, d'une **interaction indirecte** si aucun ne connaît l'autre. L'interaction peut ne faire intervenir que deux correspondants (dite **interaction point à point**), ou bien être une **interaction pluraliste**. Ce peut être une **interaction symétrique** (chacun des correspondants a la même fonction) ou bien une **interaction asymétrique** (l'un est spécialisé : maître, initiateur...). En particulier dans l'interaction pluraliste, il peut y avoir un correspondant qui a la fonction d'émetteur alors que tous les autres sont récepteurs, ou bien l'inverse. Ce peut être une **interaction synchrone** si pour chaque correspondant l'interaction n'est finie que quand tous les correspondants ont interagi, ou bien une **interaction asynchrone** si certains correspondants peuvent considérer l'interaction comme finie alors que d'autre n'ont pas encore interagi. En ne considérant que le point de vue local d'une interaction, une primitive servant à réaliser une interaction peut être une **primitive bloquante** si le processus qui l'exécute est bloqué jusqu'à ce que l'interaction, du point de vue local, soit terminée, ou bien une **primitive non-bloquante** si le processus n'est pas bloqué mais doit tester la complétion de la partie *locale* de l'interaction et par exemple ne pas réutiliser le tampon de communication tant que la complétion de l'interaction n'est pas signalée. Enfin ce peut être une **interaction déterministe** si on sait toujours quelle interaction sera exécutée ou bien une **interaction**

indéterministe si l'interaction exécutée dépend soit d'un choix à l'exécution indéterministe, soit d'un état de l'exécution (état d'une variable d'exécution, possibilité d'effectuer une interaction sans délai, qualification des interaction possibles, priorités dynamiques de celles-ci...).

Quatre grandes classes d'interaction sont présentées ici. Ce choix est représentatif des notions d'interactions, présentes dans les exécutifs parallèles, qui ont des implantations très efficaces sur machines à mémoire distribuée. Nous n'aborderons pas ici deux autres grandes classes d'interactions moins efficaces sur ces machines, la mémoire globale associative, dont le modèle est illustré par exemple par Linda [Carriero & Gelernter 1989], et les mémoires virtuellement partagées que certains supports exécutifs mettent en œuvre. Nous recommandons la lecture de [Bal et al. 1989, Andrews & Schneider 1983] pour une présentation plus approfondie de celles-ci.

Modèle à base de processus échangeant des messages de façon synchrone Cette interaction est proche de ce qui se passe physiquement entre processeurs : des messages (paquets de données) sont échangés entre les processus. Un simple multiplexage des communications est instauré pour autoriser le partage des ressources de communication par plusieurs processus : chaque message est précédé d'une information permettant de savoir à quel processus il est destiné.

L'interaction est habituellement asymétrique : un processus reçoit ce qu'un autre lui envoie. Elle est synchrone : un processus qui doit recevoir une donnée réserve un espace mémoire pour cela et se bloque tant que l'émetteur n'a pas émis. Le processus émetteur émet une donnée résidant à une adresse mémoire définie, et se bloque tant que le receveur n'est pas prêt. Un exemple typique est la communication de type *Synchronous* de MPI [Message Passing Interface Forum 1994, Gropp et al. 1995, Message Passing Interface Forum 1995]. Il y a donc synchronisation des participants, avec le risque d'un interblocage s'ils ne sont pas d'accord sur l'ordre des communications. Etant proche de la machine physique, la réalisation demande un minimum de surcoût de gestion et est donc généralement performante.

En cas d'hétérogénéité de format de données entre les processeurs, le typage logique des données transmises par échange de message doit être connu de chaque intervenant. MPI propose un typage sous la forme d'une « carte mémoire » : on décrit d'une façon complexe et générale le type de chaque donnée devant faire partie du message, en identifiant sa position en mémoire relativement à une adresse de base. Le type des données d'un message peut être utilisé plusieurs fois et servir à construire de nouveaux types. Il est donc établi préalablement à l'envoi d'un message. Les données d'un message ne sont pas forcément contiguës en mémoire ; le mécanisme de typage des données de MPI permet, à lui seul, de réagencer les données lors du transfert, en changeant leur agencement chez le destinataire par rapport à ce qu'il était chez l'émetteur.

En général, les communications dans ce mode d'interaction sont point à point et les correspondants sont parfaitement connus. Trois formes d'extensions sont cependant courantes :

- les communications collectives. Il s'agit de la diffusion (un processus émet vers plusieurs), de la concentration (plusieurs processus émettent vers un seul), de l'échange total (chaque processus émets vers tous les autres) ; avec deux formes de variantes : les limitations de portée (restriction de l'interaction à certains processus), la personnalisation (les données émises dépendent du processus destinataire). Des réductions sont aussi possibles (ie . un processus reçoit plusieurs messages et les « réduit » à l'aide d'un opérateur quelconque comme +, max ...). Dans une communication collective, les correspondants sont habituellement parfaitement connus. Si la géométrie du réseau d'interconnexion est connue, il est généralement possible d'avoir une implantation efficace de ces opérateurs. MPI par exemple propose un grand nombre d'opérations collectives, qui ont la particularité d'être symétriques.
- l'indéterminisme. Le processus récepteur accepte de recevoir n'importe quel message, provenant de n'importe quel autre processus, de façon soit complètement indéterministe (et même

équitable), soit sélective en fonction de l'état interne du processus ou bien du contenu du message. L'indéterminisme est généralement associé à une construction semblable au ALT d'Occam [Pountain 1986]. MPI propose un indéterminisme sans sélection, pour toutes les communications point à point : la réception depuis n'importe quel émetteur (*ANY_SOURCE*). Notons que l'indéterminisme n'est généralement réalisé qu'en réception, car l'existence d'un indéterminisme en réception et émission complique singulièrement la réalisation du protocole de communication.

- le filtrage. Certaines communications peuvent être privilégiées par rapport aux autres. Dans MPI par exemple, il est possible de définir des plans de communication (*communicators*), qui permettent d'identifier et de séparer plusieurs canaux de communication (qui virtualisent un lien physique). L'état d'un canal de communication est indépendant de celui des autres ; certains canaux peuvent être passants alors que d'autres sont bloquants.

Des formes non-bloquantes d'émission ou de réception sont tout à fait possibles ; cependant elles ne changent pas l'aspect synchrone de la communication : ce sont de simple optimisations destinées à autoriser la communication le plut tôt possible, en concurrence de l'exécution du processus.

Ce modèle d'interaction est généralement employé dans les paradigmes de parallélisme de données ou pipeline, car sa structure statique convient bien.

Modèle à base de processus échangeant des messages par boîtes aux lettres L'échange synchrone de messages n'est pas souple, car l'on ne peut pas filtrer les messages, c'est à dire changer l'ordre de réception par rapport à celui d'émission, *pour une paire de processus* (ou plutôt un canal de communication) donnée. Pour obtenir ce type de comportement, il faut soit multiplier les processus (et s'occuper des synchronisations associées), soit multiplier les canaux de communication et user de l'indéterminisme... Ces deux solutions, quoiqu'efficientes, ne sont pas faciles à concevoir et prouver. D'autre part, nous avons vu que l'ordre de communications synchrones doit être connu à l'avance et respecté sous peine de générer des interblocages. Enfin, dans ce mode synchrone il n'est pas facile d'ajuster les différences de vitesses entre les processus. La synchronisation, particulièrement forte, peut être à l'origine de nombreux délais d'attente. Ces trois faits conduisent à introduire des tampons dans la communication, nommés « boîtes aux lettres ». Les processus ne sont plus alors synchronisés fortement : le processus émetteur remplit un tampon dans la boîte aux lettres et peut vaquer à d'autres occupations ; le processus récepteur peut tester la présence d'un message particulier dans la boîte (généralement identifié par une marque particulière, son *étiquette*), attendre ce message, le transférer dans son espace mémoire. Le filtrage, du point de vue du programmeur, est donc très simple à utiliser.

Comme les processus passent par une boîte aux lettres, l'interaction n'est plus forcément directe : le processus émetteur peut même avoir cessé d'exister lorsque le récepteur lit le message. Une même boîte aux lettre peut être partagée par plusieurs processus, permettant une nouvelle forme d'indéterminisme (sur le récepteur). Les processus communicants peuvent exécuter des algorithmes différents, puisque la synchronisation est très lâche. Les interactions par boîtes aux lettres sont asynchrones. Elles peuvent prendre les mêmes formes point à point ou collectives, bloquantes ou non, déterministes ou non, que l'échange de message synchrone. Cependant, le filtrage des messages peut aboutir à ne plus respecter l'ordre d'émission.

En contrepartie de cette souplesse de fonctionnement, le coût de gestion des messages est plus élevé puisque qu'ils ne sont pas consommés directement, mais copiés temporairement. Par exemple PVM[1] propose un emballage des données dans des tampons puis l'émission des tampons, et le déballage depuis les tampons de la boîte aux lettres. Le type des données est donc spécifié pour chaque donnée, à chaque emballage / déballage. Cette manipulation est plus lourde que celle proposée par

1. PVM est décrit en détail au chapitre suivant

MPI, mais plus souple car elle ne nécessite pas la construction d'une carte mémoire préalable ; on spécifie juste une liste de types de données. Comme dans MPI, les données peuvent être discontiguës en mémoire.

D'autre part un problème d'allocation de ressources pour la boîte aux lettres se pose, car elle représente un espace de stockage difficilement borné. Une approche consiste à retomber dans un mode synchrone si la boîte est pleine - avec le danger de générer des interblocages ; une autre solution est de considérer cette condition comme une erreur fatale (c'est l'approche retenue par PVM).

Ce modèle est implanté par un grand nombre de noyaux de communication, Express [Flower et al. 1991, Parasoft Corporation 1992, Parasoft Corporation 1988], CHIMP [Edinburgh Parallel Computing Centre 1991, Edinburgh Parallel Computing Centre 1992], LAM [Burns et al. 1994], P4 [Butler & Lusk 1994, Butler & Lusk 1992], PICL [Geist et al. 1992], CHAMELEON [Gropp & Smith 1993], ZIPCODE [wei H. Lehman 1993], NX [Pierce 1988]... Les plus populaires sont cependant PVM et MPI. Ce modèle est suffisamment souple pour permettre aisément n'importe quel paradigme de parallélisme.

Modèle client-serveur Les deux modèles à échange de message ne couplent pas création de processus et communication. Généralement, ces deux modèles sont complétés par une primitive de création de processus, soit locale, soit à distance, sans transmission d'information (ou avec une forme réduite ou atypique). Par exemple PVM propose une primitive *pvm_spawn* pour créer un ou plusieurs nouveaux processus à distance, mais n'autorise pas la transmission d'un message durant la création. A l'inverse, le modèle client- serveur couple généralement activation / terminaison de processus et communication / synchronisation.

Le modèle client-serveur est en quelque sorte lui aussi fortement basé sur les mécanismes physiques des machines parallèles. En effet, lors de l'arrivée d'un message, une démarche usuelle est de générer une interruption, qui active un traitement particulier, le gestionnaire ou **traitant d'interruption**. Ce traitant a en charge d'intégrer le message dans le déroulement de l'application. Par exemple, dans une interaction par boîtes aux lettre, le traitant stocke le message dans une boîte spécifique. Le modèle d'interaction client-serveur remplace ce traitant statique, qui ne fait que stocker le message, en un traitant dynamique, qui active ou génère un nouveau processus.

Un exemple particulier est celui des **messages actifs** (*active messages*, [von Eicken et al. 1992, von Eicken & Culler 1992]). Un tel message contient en entête l'adresse d'une fonction à exécuter, qui va intégrer le reste du message dans le déroulement de l'application. Cette fonction est très proche d'un véritable traitant d'interruption ; elle interromps l'activité principale. Pour cette raison, peu d'actions sont permises à cette fonction : elle peut stocker le message à une adresse prédéterminée, calculer un résultat partiel à partir du message (comme dans une étape de réduction), réveiller un processus, mais généralement elle ne peut pas créer un nouveau processus ou envoyer un message de réponse, car cela nécessite une propriété complexe de réentrance[2] , pas toujours disponible.

Pour cette raison, les messages actifs ne sont pas destinés au programmeur d'applications. Cependant, d'autres modèles de plus haut niveau lui permettent d'effectuer une interaction dynamique (donc avec création de processus). Nous distinguons deux formes d'interaction client-serveur : l'**appel de procédure à distance** (*Remote Procedure Call*) et le **rendez-vous**. Le client demande une fonctionnalité au serveur. Celui-ci peut l'exécuter par la création d'un nouveau processus (c'est ce que nous qualifions ici de RPC, pour distinguer cet aspect du suivant) ou par la synchronisation avec le processus serveur (ce que nous nommons un rendez-vous). Dans cette dernière forme, un opérateur particulier, par exemple la primitive *accept* du langage ADA, permet cette synchronisation.

2. Ce terme est défini au prochain chapitre

La communication est bidirectionnelle : le client émet une requête et reçoit une réponse. L'interaction est donc synchrone. Elle n'est cependant pas forcément bloquante pour le client. Elle est par essence asymétrique : le client et le serveur ont des rôles distincts. La spécification des types des données est réalisée comme pour un appel de procédure local : par passage de paramètres, par valeur. Il est ainsi possible de réaliser des compilateurs chargés de vérifier les types des données, ce qui est un grand avantage vis à vis de l'échange de message. Le compilateur génère quatre fonctions, dites **talon** (*stub*) chargées d'emballer ou de déballer les données dans les messages. Deux talons sont créés pour le client, deux autres pour le serveur, permettant de traiter de façon transparente chaque message de requête et de réponse. Les données manipulées par les talons peuvent être discontiguës en mémoire, et même d'un type construit comme une structure, voire une arborescence.

L'indéterminisme (ou l'exclusion) sont possibles grâce à un mécanisme de **gardes** : un serveur peut bloquer certaines fonctionnalité si son état est inadéquat ou même si les paramètres de la requête sont inappropriés. L'interaction est en général directe, mais peut devenir indirecte par l'introduction d'un serveur de noms, intermédiaire entre le client et le véritable serveur. Dans la quasi-totalité des implantations, c'est une interaction point-à-point.

Pour le programmeur, l'interaction client-serveur permet de conserver une forme procédurale, bien connue, tout en profitant de la distribution. Enfin l'appel de procédure à distance, tel que nous l'avons défini ici, favorise le parallélisme en générant de nouveaux processus.

Ce modèle est implanté notamment par ADA [Ledgard 1982, Barnes 1991] et Concurrent C [Gehani & Roome 1986] pour le mode rendez-vous et Distributed Processes [Seevers et al. 1992] et DCE [Shirley 1992, Rosenberry & Teague 1993] pour le mode RPC. SR [Olsson et al. 1992, Coffin & Olsson 1989] implante les deux notions. Le lecteur intéressé par la notion d'appel de procédure à distance consultera avec profit les articles [Birrell & Nelson 1984, Sun Microsystems Inc. 1988, Sun Microsystems Inc. 1986, Weihl 1989, Bershad et al. 1990, Branstetter et al. 1991]. Le modèle client-serveur convient particulièrement au paradigme de parallélisme de contrôle.

Une forme réduite de client-serveur est offerte par l'**exécution de service à distance** (*Remote Service Request, RSR*). Ce mécanisme découple requête et réponse. Chaque message est traité comme une interaction client-serveur, mais « sans retour ». Une réponse est traitée comme une requête, conduisant à la création d'un nouveau processus, chargé d'intégrer la réponse chez l'appelant. L'appelant spécifie lui-même, dans sa requête initiale, l'identité de ce processus. L'avantage de ce mode est de permettre une économie de messages (cas où il n'y a pas de réponse utile), et de relâcher la synchronisation dans le programme (le « retour » peut être effectué zéro, une ou plusieurs fois, vers des processus éventuellement différents de celui qui a généré la requête initiale). L'inconvénient est double : premièrement l'intégration de la réponse est plus complexe car elle introduit un nouveau (troisième) processus, et nécessite une synchronisation entre ce processus et celui qui a envoyé la requête initiale, deuxièmement, la structure du programme n'est pas aussi clairement définie que dans le modèle procédural stricte.

Cette forme réduite permet une approche analogue aux boîtes aux lettres, qui consiste à considérer que chaque message est un événement qui sera temporairement stocké dans un réceptacle virtuel et qui à un moment donné générera un nouveau processus ayant comme argument le message.

Dans ce modèle, que nous nommons **modèle à événements générant des processus**, on considère comme acquis le traitement d'un message, mais on veut pouvoir agir sur l'ordonnancement des traitements (et donc des messages). Le point clé est donc le choix du moment d'activation de l'événement. Les performances d'un tel système dépendent très fortement de la réalisation de l'ordonnanceur des événements (centralisé, distribué) et des politiques d'ordonnancement qui sont appliquées. Ce modèle est implanté dans Charm [Univ. Illinois 1992, Kale & Krishnan 1993, Kale 1994] et Cilk [Blumofe et al. 1995] notamment. Dans Cilk la création de processus est suivie d'une attente de disponibilité de valeurs. Ce peut donc être une interaction pluraliste (un ET de messages indépendants) qui déclenche

le processus.

Si les interactions possibles entre processus sont réduites à la seule opération de création d'un nouveau processus, le graphe de dépendance de l'application est particulièrement simple : il ne comporte pas de cycle. En contrepartie la localité d'accès aux données est difficile à assurer : les données « migrent » d'un processus à l'autre. Ce modèle correspond à une extension du paradigme de parallélisme pipeline, nommée **parallélisme flot de données**, dans laquelle les traitements effectués dépendent des données.

Modèle à base d'accès mémoire à distance Dans une interaction par échange de messages, les processus qui communiquent existent préalablement et sont équivalents. Dans l'interaction client-serveur, le processus client active ou génère le processus serveur. Une troisième grande forme est l'accès mémoire à distance, dans laquelle il n'y a qu'un seul processus impliqué. C'est ce que nous appelons une **interaction unilatérale** (*one-sided*).

L'accès mémoire à distance, c'est la faculté pour un processus de lire, écrire ou modifier un zone mémoire située sur un autre nœud. Cette interaction peut être aidée par un mécanisme matériel, analogue à un *DMA, Direct Memory Access*, à travers le réseau d'interconnexion, c'est à dire à une copie directe de mémoire à mémoire, sans intervention des processeurs de calcul. Cette forme d'interaction est présente par exemple dans ParX (voir plus loin).

Elle est en quelque sorte synchrone (les accès se font dans l'ordre où ils ont été émis, pour une paire de processeurs donnés). Une forme pluraliste existe, dans laquelle plusieurs processus accèdent collectivement à la mémoire, avec un certain ordre, par exemple chacun à son tour, ou bien chacun accédant une zone de données différentes. Un fonctionnement indéterministe peut être obtenu, par exemple quand plusieurs processus écrivent au même emplacement mémoire. Le filtrage et la sélection sont aussi possibles à travers des opérateurs atomiques de lecture et modification de la mémoire distante, qui vont agir sur des verrous distants. Par essence, les primitives qui la mettent en œuvre sont généralement non- bloquantes. Une sorte de « compteur de modifications » peut être associé à la zone mémoire distante, permettant de maintenir une certaine cohérence dans les accès.

Cette forme est proche du matériel, et à notre connaissance, peut proposée au programmeur d'application. Elle est généralement dédiée à une architecture homogène (donc les types des données ne sont pas explicités), les données étant identifiées de façon contiguë. Le draft MPI- 2[3] propose cependant des données discontiguës, dans une architecture hétérogène, au programmeur d'application. Dans ce modèle d'interaction, comme dans ceux à échange de messages, la création de processus est faite d'une façon distincte de la communication / synchronisation.

Conclusion Il existe plusieurs formes d'interactions entre les processus. Nous en avons donné quelques
exemples, généralement proches de la machine physique à mémoire distribuée ; nous avons mis en lumière quelques-unes de leurs caractéristiques principales. Notre propos n'est pas de promouvoir une forme plutôt qu'une autre, mais simplement de faire remarquer la grande diversité de conception possible dans l'expression du parallélisme.

2.2.4 Les supports exécutifs pour le parallélisme

Nous avons dans la section précédente identifié les principaux modèles d'expression du parallélisme (sur machines à mémoire distribuée). Ces modèles sont proches de la machine physique : les

3. Voir plus loin, dans le chapitre de comparaison.

concepts manipulés sont les flots d'instructions, les périphériques de communication, les données et leur stockage en mémoire. Leur réalisation sur une machine nue est donc relativement directe. D'autre part, ces concepts sont très proches de ceux proposés par les systèmes d'exploitation « classiques » (à la Unix BSD par exemple). Une approche naturelle est donc d'englober le support du parallélisme dans le système d'exploitation de chaque nœud de la machine, donnant de ce fait un nouveau système d'exploitation, spécialisé, nommé « parallèle ». Cette approche a été utilisée, comme nous allons le voir, pour un certains nombre de machines parallèles intégrées « d'ancienne génération ».

Cependant, les machines parallèles formées de réseaux de stations, ou bien intégrant des nœuds ordinaires autour d'un réseau rapide (ce que nous appelons machine parallèle intégrée « de nouvelle génération »), ont déjà un système d'exploitation classique présent sur chaque nœud. Le support d'exécution du parallélisme est alors une couche supplémentaire qui complémente le système classique : soit en permettant une utilisation directe des périphériques spéciaux que sont les réseaux rapides, soit en introduisant des protocoles pour la réalisation d'applications distribuées, soit en définissant une interface portable pour le support du parallélisme. Dans la mesure où nous ne considérons pas les machines à mémoire partagée, nous ne nous intéressons ici qu'aux supports d'exécution parallèle pour *machines à mémoire distribuée.*

2.2.4.1 Les noyaux de systèmes parallèles

Un certain nombre de noyaux de systèmes parallèles ont été proposés, la plupart pour une machine ou une catégorie de machines spécifique. Ces noyaux sont nécessaires à l'exploitation de la machine ; ils incluent le support du parallélisme proprement dit, c'est à dire implantent un ou plusieurs des modèles d'expression du parallélisme que nous avons présentés. Cependant ces noyaux offrent une gestion complète de la machine, avec un intérêt en particulier pour :

- le démarrage et la configuration de la machine,
- le partitionnement physique et temporel de la machine entre plusieurs utilisateurs ou programmes,
- les entrées-sorties avec le monde extérieur,
- la gestion d'espace mémoire virtuel,
- la gestion d'un espace de stockage permanent ou temporaire...

A un niveau plus haut, ces systèmes proposent souvent une régulation de charge par placement automatique des nouveaux processus.

Des exemples de tels systèmes sont donnés par ParX [T.Muntean 1994, T.Muntean et al. 1993, Tsobgny 1991], Peace [Schroeder-Preikschat 1990], Helios [N.H.Gernett 1987], Trollius [M.Braner 1988]. Nous allons maintenant détailler ParX qui est à la fois représentatif de ces systèmes et présente un concept de généricité intéressant.

Un exemple : ParX ParX est le noyau « parallèle » du système ParOs, développé à Grenoble. Le système ParOs est un système générique, offrant plusieurs sous-systèmes différents à l'utilisateur, de façon à ce qu'il puisse choisir le modèle de programmation qui lui est le plus approprié (PCTE, X/OPEN, interface PVM, mémoire virtuellement partagée...). Chaque sous- système est construit sur le micro-noyau dit π-nucleus, qui met en œuvre les politiques d'allocation des ressources de base (processeurs, mémoire, etc...) et garde un contrôle sur les entités de base du système (processus et objets de communication). Ce micro- noyau offre un mécanisme générique pour la construction des protocoles utilisés par les divers sous-systèmes (diffusion, client-serveur etc...). Chaque protocole peut coexister et fonctionner en concurrence sans influence sur les autres.

ParX offre à la fois un contrôle sur la machine physique (par exemple il est possible de la partitionner en *clusters*, de reconfigurer le réseau physique dans certains cas...), et une vision virtualisée de celle-ci. La *Ptâche* est une machine virtuelle, regroupant dans une même entité administrative des *tâches* (processeurs virtuels, notion d'espace d'adressage), chaque tâche hébergeant plusieurs flots d'exécution légers (ces flots d'exécution sont matériellement supportés par les processeurs *Transputers* qui sont la cible de ce système).

La machine cible ayant un réseau de proche en proche, une fonction importante de ParX est la réalisation du routage des messages dans le réseau, sans introduire d'interblocage. Au dessus de cette fonctionnalité sont réalisés différents protocoles de communication. Par nécessité sur des nœuds disposant de peu de mémoire, la couche de routage n'inclut qu'un nombre restreint de tampons de communication (un par lien de communication physique entrant ou sortant, plus un par protocole supporté). La condition de non-interblocage du routeur implique donc un comportement particulier des protocoles, qui doivent accepter inconditionnellement tout message en un temps fini, et donc ne pas générer d'interblocage au niveau même du protocole. Le principe d'implantation générique des protocoles est d'assurer un contrôle de flux tel qu'il y ait toujours suffisamment de tampons libres au niveau du protocole pour qu'aucun interblocage ne puisse se produire. Différents protocoles ont été implantés, comme par exemple l'accès mémoire à distance, l'échange de message synchrone, le rendez-vous client-serveur etc...

Notons que dans ParX, chaque arrivée d'un message génère une interruption, qui active la couche de routage et par la suite le protocole indiqué dans l'entête du message. L'intégration des messages dans le déroulement de l'application se fait donc de façon forcée. Lorsqu'un processus léger se bloque, le matériel active de façon automatique un autre processus léger prêt. Cependant, l'ordonnancement des processus légers peut être contrôlé au niveau de ParX, à l'aide d'une notion de priorité d'exécution.

Les noyaux de systèmes distribués et parallèles A coté de ces noyaux de systèmes parallèles, des micro-noyaux de systèmes distribués, comme Chorus [L.Albinson et al. 1991, B.Herrmann & L.Philippe 1992], Mach [Accetta et al. 1986], Amoeba [Mullender et al. 1990], sont maintenant couramment utilisés dans des machines parallèles. Ces micro-noyaux offrent la quintessence d'un noyau parallèle :

- le support de flots d'exécution « légers »,
- le support d'espaces d'adressage et de protection (« tâches »),
- un mécanisme de communication entre espaces d'adressage. Ce mécanisme est généralement basé sur l'envoi de messages synchrone.

Les fonctionnalités supplémentaires d'un système d'exploitation (entrées-sorties, systèmes de fichiers etc...) sont regroupées dans des tâches particulières (dites serveurs). Ces micro- noyaux ont investi les machines parallèles par l'intermédiaire des applications de contrôle- commande. Par exemple, un noyau « nu » est présent sur les nœuds de base. Il offre les fonctionnalités essentielles pour la réalisation de la partie commande. Des nœuds particuliers, dotés de périphériques spécifiques, sont utilisés par exemple comme serveurs de fichiers, comme terminal de contrôle, etc... Le même noyau de système, complété par les services spécialisés correspondants, réside sur ces nœuds.

Le paradigme de parallélisme principalement employé avec ces noyaux est celui d'acteurs. La liaison entre le calcul haute performance et ces noyaux de système à la fois distribués et parallèles ne nous semble pas encore réalisée. Nous ne détaillons donc pas plus leurs mécanismes, sachant qu'ils sont assez semblables à ceux des noyaux de systèmes parallèles que nous avons présentés, en tout cas pour la partie parallélisme.

2.2.4.2 L'utilisation directe d'une machine parallèle

La section précédente présentait les noyaux de systèmes construits d'une façon ad-hoc pour une machine ou classe de machines. Cependant, élaborer un tel noyau, ou même simplement le porter, est une tâche longue, difficile et donc coûteuse. L'approche retenue par beaucoup de constructeurs de machines parallèles est l'utilisation d'un système d'exploitation classique, en le complétant par les mécanismes nécessaires pour le support du parallélisme.

Ces mécanismes sont en priorité la gestion d'un adaptateur réseau rapide. Cet adaptateur réseau peut être accédé par plusieurs processus ; il faut alors gérer l'accès à l'adaptateur, et cela est du ressort du système d'exploitation. Cependant, certains paradigmes de parallélisme (comme le parallélisme de données) ne supposent qu'un seul processus par nœud de la machine. L'accès à l'adaptateur ne pose alors pas de problème. Une approche effectivement employée pour le support du parallélisme est donc de rendre l'adaptateur réseau rapide directement accessible à un processus unique, qui peut alors implanter n'importe quel protocole de communication. Comme cette tâche reste encore complexe, le constructeur peut fournir un protocole par défaut, comme par exemple l'échange de message. C'est l'approche retenue par IBM pour l'adaptateur TB0 : une bibliothèque, *CSS-CI*, est fournie avec la machine SP1, qui par ailleurs possède sur chaque nœud un noyau de système Aix tout à fait standard.

Notons que la bibliothèque, ou le protocole de communication, défini dans ce cadre est souvent spécifique au constructeur - qui cherche entre autres à optimiser l'utilisation de son matériel, et propose donc des opérateurs spécifiques et efficaces. Les mécanismes usuels du système d'exploitation classique sont employés pour la réalisation des fonctions, autres que la communication, nécessaire au support du parallélisme : gestion des ressources, gestion de processus, observation et contrôle.

2.2.4.3 Les systèmes d'exploitation réseau et le parallélisme

Les systèmes d'exploitation ont rapidement intégré des extensions pour la mise en réseau des machines. Ces extensions s'occupent de différents niveaux dans la communication, allant des trames échangées sur le médium physique, jusqu'aux protocoles utilisés pour échanger des informations entre des applications en milieu hétérogène. Les différentes couches de protocoles sont habituellement identifiées selon la norme ISO.

Ces systèmes d'exploitation réseau intègrent des mécanismes de distribution d'application, basés soit sur l'échange de messages (*sockets* en mode soit UDP [Postel 1981*b*], soit TCP [Postel 1981*a*], soit sur le client-serveur (*Distributed Computing Environment*, DCE [O'Reilly & Associates 1991], ou *Common Request Broker Architecture*, CORBA [Otte et al. 1996], en sont l'exemple type). Sont-ils pour autant de bon candidats pour l'exécution d'applications parallèles ?

La réponse est généralement non. En effet, ces systèmes ciblent des applications distribuées, généralement sur des réseaux étendus. L'efficacité de la communication a alors moins d'importance que la sécurité ou la souplesse de fonctionnement. Par exemple, dans DCE ou CORBA, l'appel de procédure à distance se fait à travers un serveur - ou *broker* - qui permet de localiser le service appelé dans le réseau. L'appel est fiabilisé par un protocole permettant une reprise automatique des erreurs, et sécurisé par l'authentification et le contrôle d'accès des intervenants. Même si (comme dans DCE ou CORBA), l'utilisation de processus légers et de réseaux rapides est possible, nous pensons que les systèmes d'exploitation réseau sont trop chargés en fonctionnalités dédiées aux applications distribuées pour être réellement exploitables dans le contexte du parallélisme.

2.2.4.4 Les bibliothèques portables pour le parallélisme

Le problème de l'approche constructeur est le manque de portabilité de l'outil de programmation. Le problème des systèmes d'exploitation réseau est la trop grande importance accordée à l'aspect distribution. Des interfaces générales et donc facilement portables ont été définies spécialement pour le support du parallélisme. Ces interfaces, dont les plus célèbres sont PVM et MPI, visent à assurer la portabilité d'une application parallèle sur une large gamme de machines. Cependant, elles sont issues des bibliothèques de communication utilisant directement une machine parallèle. Elles capitalisent sur le système d'exploitation, commun ou très semblable, des différentes machines cibles pour réaliser leur support du parallélisme.

En particulier, elles réutilisent les mécanismes usuels de support des processus présents dans les systèmes d'exploitation usuels : les processus sont lourds et relativement statiques. D'autre part, elles capitalisent sur la grande quantités de mémoire maintenant disponible sur chaque nœud pour effectuer un tamponnage important des messages. Ce tamponnage est à la fois un remède contre la latence des communications, mais aussi apporte une grande souplesse dans le filtrage est messages, et permet donc de n'utiliser qu'un seul flot d'exécution pour traiter une multitude de messages. Ces bibliothèques offrent un environnement de programmation souple et généralement agréable, permettant souvent l'observation du comportement de l'application, l'évaluation de performances, le déverminage. Nous présentons plus loin en détail PVM ; nous laissons le lecteur se rapporter aux documents de référence sur MPI [Message Passing Interface Forum 1994, Gropp et al. 1995, Message Passing Interface Forum 1995].

2.2.5 Conclusion : caractéristiques générales d'un support exécutif pour applications parallèles

En conclusion de notre étude, un support exécutif pour applications parallèles sur machine à mémoire distribuée doit être à même de :

- gérer les ressources de la machine physique, en produire une vision virtuelle,
- gérer des séquences de calcul (processus),
- gérer des séquences de communication (canaux de communication),
- réaliser, à l'aide des séquences de communication, diverses interactions entre les séquences de calcul,
- permettre l'observation et le contrôle de l'application parallèle.

Nous pouvons distinguer trois grandes formes d'interaction efficace pour des machines à mémoire distribuée :

- l'interaction par échange de messages (synchrones ou par boîtes aux lettres) entre entités prédéfinies,
- l'interaction par échange d'information durant la création ou destruction d'activités (génération de processus ou client-serveur).
- l'interaction unilatérale par accès mémoire à distance.

Nous avons vu que, aussi bien au niveau des processus que des canaux de communication, trois grandes caractéristiques prédominaient :

- l'aspect dynamique ou non de la gestion,
- le coût de l'exécution (lourd ou léger),
- le contrôle de l'ordonnancement possible.

Nous avons détaillé quelques organisations de supports exécutifs, selon qu'ils définissent un nouveau système d'exploitation :

- les noyaux de systèmes parallèles définissent un noyau de système dédié au parallélisme
- les noyaux de systèmes distribués et parallèles nous semblent une très bonne approche, mais ne sont pas couramment disponibles,

ou qu'ils complémente un système d'exploitation « classique » par un support spécifique pour le parallélisme :

- l'utilisation directe d'un périphérique de réseau rapide n'est pas portable,
- les « couches réseau » des systèmes d'exploitation sont trop orientées vers le support des applications réparties,
- les bibliothèques de communication dédiées au parallélisme sont portables et spécialement adaptées au parallélisme, mais n'offrent pas, pour les plus standards, la notion de processus légers, dynamiques.

Nous allons maintenant nous intéresser à un cadre de travail particulier, et vérifier l'adéquation des concepts présentés et des exécutifs existants à la programmation d'applications parallèles dans ce cadre. Nous montrerons qu'il est nécessaire de concevoir un support exécutif particulier pour cela.

2.3 Les applications parallèles *irrégulières*

Dans cette section nous précisons le domaine d'application dans lequel a été réalisé ce travail, à savoir les applications parallèles irrégulières. Nous commençons par définir ce terme, puis nous montrons la nécessité d'un exécutif spécialisé pour de telles applications.

2.3.1 Régularité et irrégularité

Une application parallèle résout un problème donné au travers d'un algorithme particulier. Cet algorithme est rendu parallèle d'une façon ou d'une autre et peut donc s'exprimer sous la forme d'un ensemble de séquences de calcul et de séquences de communications, ainsi que d'un graphe (partiel) de dépendance entre ces séquences. Notre but ici n'est pas de décrire comment un tel algorithme parallèle peut être obtenu, soit à partir d'un algorithme séquentiel ou directement à partir d'une formulation mathématique ; mais seulement de remarquer qu'une « bonne » parallélisation nécessite d'assurer un grain de calcul et de communication adapté à la machine cible, d'équilibrer la charge et de minimiser les communications. Ces trois points ont été précisés précédemment.

Ceci dit nous allons classer les applications parallèles selon leur comportement. Nous appellerons **application parallèle régulière** une application dont on peut prédire son comportement : si l'on peut prédire à l'avance, et ce quelque soit les données du problème particulier que l'on cherche à résoudre, à la fois les différentes séquences dont l'application sera composée, les relations de dépendance entre ces séquences et aussi les « volumes » de ces séquences de calcul et de communication.

Le fait de pouvoir prédire ces informations permet d'établir un **ordonnancement** (une détermination du site d'exécution et de la date d'exécution de chaque séquence) que l'on juge « bon » - il n'est probablement pas optimal mais il est suffisamment proche de l'optimal pour que l'on s'en contente - ce qui induit en particulier la possibilité de *sérialiser* les séquences sur chaque processeur. Une autre caractéristique des applications parallèles régulières est que leur expression est facilitée par cette connaissance préalable de leur comportement : dans la mesure où l'on sait qui communique avec qui et quand, on peut utiliser des échanges de messages synchrones, ou bien des communications

collectives *synchronisantes* comme par exemple une diffusion telle que définie dans le standard MPI. La dernière caractéristique que nous retiendrons des applications parallèles régulières est que cette connaissance préalable nous permet de gérer les différentes séquences de façon *statique*, c'est à dire de prévoir leur création préalablement à l'exécution.

Ces trois caractéristiques des applications parallèles régulières, la sérialisation, la synchronisation et l'aspect statique des séquences induisent une expression et un support d'exécution relativement simples. Une expression typique d'une application parallèle régulière est ainsi celle d'une collection de processus communicants par échanges synchrones de messages. Notons que cette expression est adéquate quelque soit la façon dont l'application a été obtenue : elle peut être le résultat d'un travail de conception manuelle, ou bien d'une parallélisation automatique d'un code séquentiel...

Un exemple d'application régulière est le calcul de l'itération de Jacobi. Le calcul se déroule comme l'itération de deux phases, d'abord l'échange de la valeur de tout point avec celles de ses voisins et le calcul d'une nouvelle valeur pour le point ; ensuite une réduction globale pour déterminer si les valeurs sont suffisamment stables pour que le calcul s'arrête. Bien que le nombre de points et de pas de calcul puisse varier d'un problème à l'autre, l'application est régulière puisque le nombre de points est connu, que les calculs sont sérialisés et que les communications se font de façon synchrone.

A contrario, une **application parallèle irrégulière** est une application pour laquelle on ne peut pas prédire le comportement indépendamment de l'instance particulière du problème que l'on cherche à résoudre. Cette impossibilité de prédiction peut avoir plusieurs causes :

- le nombre de séquences utilisées est variable en fonction du problème visé et de sa « complexité »,
- il y a indéterminisme dans les dépendances entre les différentes séquences,
- les volumes des différentes séquences sont variables, en particulier parce que le grain est difficilement contrôlable et peut donc être très fin.

Ces caractéristiques particulières des applications parallèles irrégulières ont trois influences majeures. Il faut pouvoir exprimer et supporter l'exécution *dynamique* d'un grand nombre de séquences de calcul ou de communication. Le grain de ces séquences n'est pas déterminé à priori et peut éventuellement être fin ; il faut donc réduire les latences au maximum et supporter une exécution à faible coût (*légère*) des séquences. Puisque l'on ne peut pas établir un ordonnancement et donc sérialiser les séquences, il faut supporter l'exécution *concurrente* sur chaque processeur de plusieurs séquences différentes.

L'usage de certaines optimisations peut rendre une application initialement régulière, irrégulière. Dans notre exemple du calcul du Jacobi, l'optimisation qui consiste à exclure certaines zones stables au fur et à mesure des calculs rends l'application irrégulière en introduisant des séquences de communication ou de calcul dynamiques. Nous voyons que le problème « calcul du Jacobi » n'est pas intrinsèquement régulier ou irrégulier, mais que la régularité est une propriété de l'algorithme associé.

Outre cette apparition de l'irrégularité dans une application initialement régulière lorsqu'on essaie de l'optimiser, l'irrégularité se trouve de façon intrinsèque dans beaucoup d'algorithmes, comme par exemple dans les domaines d'application que sont le calcul formel, l'analyse combinatoire, la météorologie, la dynamique moléculaire, la génétique... De telles applications ont toutes en commun que l'on est incapable de prédire leur comportement à partir d'une caractérisation simple des données du problème.

Une troisième forme d'irrégularité est introduite lorsque l'on combine plusieurs algorithmes, même réguliers. Par exemple, une application de météorologie réalisée comme l'interaction d'un modèle d'océan et d'un modèle de circulation de l'air, même si ces modèles sont réguliers, peut in-

troduire une grande irrégularité dans le comportement de l'application, due à l'interaction des deux algorithmes.

2.3.2 Motivations pour un nouveau support d'exécution parallèle

Les applications qui nous intéressent ici sont des applications parallèles irrégulières. Nous venons de voir que le support de ces applications nécessite de permettre dynamiquement l'exécution légère d'un grand nombre de séquences concurremment. Les noyaux de systèmes parallèles offrent justement des processus légers et des mécanismes de communication proches de la machine. Ils permettent donc cette expression dynamique d'un grand nombre de séquences concurrentes, à faible coût.

Cependant, nous ne ciblons pas une machine parallèle d'ancienne génération, pourvue d'un tel noyau de système parallèle. Nous voulons réaliser un support d'exécution qui soit portable sur une large gamme de machines, allant des réseaux de stations aux machines parallèles intégrées de nouvelle génération (dont les nœuds sont des stations classiques, avec un système d'exploitation « classique »). Nous ne voulons pas redéfinir un n-ième noyau de système parallèle pour ces machines, puisque ce n'est pas viable économiquement. Nous ne pouvons pas non plus utiliser les nouveaux systèmes d'exploitation distribués et parallèles, qui posent les mêmes problèmes de disponibilité et de portabilité, actuellement.

Nous voulons par contre profiter de l'existence d'un système d'exploitation assez complet sur chaque nœud, profiter des avantages de la standardisation qu'il nous apporte, quitte à le compléter par un support pour le parallélisme adéquat. Cette approche est aussi celle des constructeurs de machines parallèles, qui offrent généralement un support pour le parallélisme comme complément du système d'exploitation classique présent sur leur machine. Cependant, ce support est trop spécifique à un constructeur pour être facilement portable.

L'approche que nous avons donc pour la réalisation de notre support portable pour applications parallèles irrégulières est l'utilisation d'une bibliothèque portable pour le parallélisme, complétant un système d'exploitation « classique » à la Unix BSD. Cependant les bibliothèques portables qui existent actuellement ne conviennent pas. En effet, nous avons vu que ces bibliothèques - comme PVM et MPI - réutilisent les mécanismes usuels des systèmes classiques pour le support des processus : les processus sont lourds. En conséquence, les applications programmées à l'aide de ces bibliothèque ne peuvent employer qu'un petit nombre de processus (quelques-uns par nœud), d'une façon très statique, et donc avec un grain de parallélisme très grand.

A l'inverse, les noyaux de systèmes parallèles utilisaient des processus légers en partie matériellement implantés dans les processeurs de l'époque. La thèse que nous défendons ici est que *l'utilisation de processus légers réalisés entièrement en logiciel, conjointement à une bibliothèque portable pour le parallélisme, au-dessus d'un système d'exploitation classique, est un choix tout à fait possible pour un support d'exécution portable pour applications parallèles irrégulières*. Cette thèse sera étayée tout au long de ce document. D'autre part nous allons introduire une réflexion particulière au projet dans lequel s'est déroulé ce travail, qui va nous mener à un modèle d'expression du parallélisme assez original.

Une alternative à l'utilisation de processus légers existe : il s'agit de la réalisation d'un automate de communication, que nous qualifions ici sous le terme d'**automate séquentiel** à base de continuations. Un tel algorithme est séquentiel, c'est à dire qu'il n'utilise qu'un seul flot d'exécution. C'est un automate, c'est à dire qu'il passe d'un état à un autre, en fonction des entrées (les communications entrantes) qui lui sont appliquées. Si une action ne peut pas être complétée sans un changement d'état, il établit une **continuation**, c'est à dire une mémorisation du minimum d'informations nécessaire à la

reprise ultérieure de l'action. Nous verrons, lors de notre présentation de PVM, un tel automate. La forme générale d'un tel automate est la suivante :

```
loop
  alt[i] < communication entrante[i] traitable >
    reprise continuation[i]
    calculs[i] et communications sortantes[i]
    établissement continuation[i]
```

FIG. 2.4 - *Automate séquentiel à base de continuations.*

Un tel automate boucle indéfiniment, sur la réception (indéterministe) d'une communication entrante, traitable - c'est à dire qui permet de faire progresser l'état de l'automate vers un nouvel état. Cette communication peut être la continuation d'un ancien traitement. Il y a alors reprise des informations permettant de continuer ce traitement ; puis le calcul d'un nouvel état pour l'automate. Si la terminaison du traitement ne peut être effectuée à ce moment (si le traitement dépend d'autres communications entrantes), une continuation est établie, sauvegardant les informations nécessaires à la poursuite du traitement. Plusieurs transitions sont ainsi possibles, la multiplicité étant symbolisée par le [i] dans l'exemple.

Un tel automate permet l'exécution concurrente d'un grand nombre d'activités, de grain éventuellement très fin. C'est donc un très bon candidat pour la réalisation d'applications parallèles irrégulières. De plus, il est directement réalisable dans un modèle de communication par échange de messages, et donc sous PVM ou MPI.

Cependant la réalisation d'un tel automate n'est pas facile. Il faut en effet être capable d'exprimer l'application parallèle sous cette forme, de vérifier la correction de cette expression, et de la maintenir au gré de l'évolution de l'application. Une telle expression est possible, soit manuellement, avec beaucoup d'attention, soit automatiquement, à l'aide d'un « langage » d'expression adapté et d'un outil de compilation adéquat. Nous voulons décharger le programmeur d'une partie de la complexité de la programmation parallèle ; d'autre part nous ne voulons pas réaliser un nouveau langage d'expression du parallélisme et son compilateur. Le choix que nous faisons est donc l'utilisation de processus légers pour une expression que nous espérons plus naturelle et plus facile du parallélisme. Notons que l'utilisation du modèle client-serveur et de processus légers, au-dessus d'une machine à mémoire distribuée, est probablement plus coûteuse que l'utilisation d'un tel automate à base de continuations, car elle introduit des mécanismes supplémentaires par rapport aux communications par échange de messages. L'un des objectifs de ce travail sera donc de quantifier le surcoût introduit.

2.4 Le support d'exécution du projet APACHE

Nous présentons ici l'approche prise par le projet APACHE, pour aider la programmation et l'évaluation d'applications parallèles irrégulières. Nous identifions en particulier la partie du projet qui concerne notre propre travail, et les interactions qu'elle possède avec les autres axes de recherche. Nous mettrons en avant les problèmes et les choix qui ont été faits pour la réalisation du support d'exécution Athapascan-0, lorsque nous aurons introduit plus précisément les mécanismes nécessaires à sa réalisation : la multiprogrammation légère et les communication à l'aide d'une bibliothèque portable. Nous compléterons donc la description du support au début du chapitre 4.

2.4.1 La problématique d'APACHE

Nous avons abordé dans les sections précédentes l'importance de la granularité sur l'efficacité d'une application parallèle. Il existe peu d'algorithmes optimaux ou presque optimaux indépendants de l'architecture de la machine cible. En conséquence, une application est développée en vue d'une machine parallèle et doit être adaptée - *portée* - pour s'exécuter efficacement sur une autre. Ce portage doit prendre en compte le changement d'architecture et les modifications de fonctionnalités offertes par le système d'exploitation, comme tout portage. Mais le point le plus délicat[4] du portage d'une application parallèle concerne l'adaptation de la granularité de calcul de l'application aux caractéristiques de la machine cible (nombre de processeurs, vitesse relative des processeurs et du réseau). Cette adaptation peut nécessiter à l'extrême une réécriture complète de l'application, ou bien seulement l'ajustement de quelques paramètres (comme le nombre de tâches, la taille des blocs de données échangés...) si la granularité a été « virtualisée » lors de la conception.

Le principal problème que rencontrent les utilisateurs de machines parallèles concerne l'évolution rapide de la technologie qui rend les machines obsolètes en peu de temps, et nécessite donc un portage incessant des applications. La programmation parallèle ne sera économiquement viable que s'il existe un environnement de programmation parallèle portable qui permette de s'adapter à la granularité de la machine cible. Cet environnement de programmation doit comporter un support d'exécution de base qui masque les problèmes de portabilité de base : principalement les différences des fonctionnalités des systèmes d'exploitation. Il s'agit donc de définir *un ensemble de fonctionnalités simples permettant la programmation parallèle indépendamment de la machine cible* ; cela constitue le premier axe d'APACHE et notre contribution personnelle au projet.

Cependant ce support d'exécution de base ne peut pas agir sur la granularité de l'application. Des fonctionnalités de régulation de charge - statique ou dynamique - sont donc nécessaires pour obtenir une portabilité *efficace* des applications. Il s'agit ici de savoir décrire un ensemble de travaux à travers une granularité variable ; c'est le deuxième axe d'APACHE.

L'application ainsi décrite devrait être capable de s'adapter automatiquement aux conditions d'exécution sur la machine cible. Le comportement d'une telle application doit être observé et ses performances analysées en vue de raffiner la description de l'application jusqu'à ce que son efficacité soit acceptable. Le troisième axe du projet APACHE concerne donc la « mise au point des performances ». La programmation d'applications parallèles est aussi rendue difficile par la concurrence présente entre chaque processus. L'environnement de programmation parallèle est donc complété par des outils de déverminage parallèle.

Enfin, le projet APACHE doit valider l'environnement de programmation parallèle ainsi créé à l'aune des applications. A travers ses compétences internes, le calcul formel, le calcul logique et la dynamique moléculaire ont été sources d'applications pour APACHE. Le groupe de travail PARAPLI, réunissant plusieurs utilisateurs de calcul haute performance, est l'ouverture privilégiée du projet en direction de la communauté du calcul parallèle française.

Les applications plus particulièrement visées par le projet APACHE sont les applications parallèles irrégulières [Gautier et al. 1995]. Pour ces applications, la parallélisation automatique ou la programmation par parallélisme de données ne sont pas particulièrement adaptés puisqu'elles ne présentent pas de structures régulières sur lesquelles appliquer ces techniques. En particulier l'échange de messages synchrones n'est pas bien adapté puisque l'on peut difficilement prévoir la quantité de données à échanger ou l'instant où la communication sera faite. En conséquence l'approche retenue

4. Il y a quelques années à peine, ce chapitre aurait mentionné la topologie des réseaux comme l'un des obstacles principaux au portage d'applications parallèles. Actuellement cependant, les noeuds sont considérés comme totalement interconnectés ; le coût de communication de n'importe quel noeud vers n'importe quel autre est identique. Cela est dû principalement aux réseaux d'interconnexion rapides maintenant disponibles.

par le projet APACHE est le parallélisme de contrôle qui considère un calcul parallèle comme un ensemble de calculs élémentaires, sur lequel est défini un ordre de dépendance partiel. Dans le but de faciliter l'expression des ces calculs élémentaires, le projet se base sur la notion de « *procédures parallèles* » : le mécanisme d'interaction de base est une extension du modèle client-serveur.

2.4.2 Athapascan-0 et Athapascan-1

Le support d'exécution du projet APACHE, nommé Athapascan, est divisé en deux grandes couches.

Le niveau Athapascan-0 [Christaller 1994*b*, Christaller 1994*a*, Briat et al. 1994, Christaller 1994*c*, Christaller et al. 1995*c*, Christaller & Castaneda-Retiz 1996] est la couche de base portable. Ce niveau permet une certaine indépendance vis à vis de la machine physique : il fournit une virtualisation de la machine parallèle en offrant des processeurs virtuels. Cependant, la machine virtuelle n'est pas déterminée de façon ad-hoc à ce niveau ; il renseigne l'utilisateur sur les caractéristiques de la machine physique et lui propose des opérateurs pour qu'il forme « sa » machine virtuelle. Ayant le contrôle de la machine virtuelle, le niveau Athapascan-0 renseigne aussi l'utilisateur sur la charge associée à chaque processeur virtuel. Il expose un parallélisme explicite qui consiste à placer des activités sur certains processeurs virtuels explicitement nommés. Il propose aussi dans une certaine mesure à l'utilisateur des possibilités pour ordonnancer les différentes activités à l'intérieur d'un processeur virtuel. Le niveau Athapascan-0 se décompose principalement en deux blocs : une interface de programmation et un exécutif support d'exécution. Le niveau Athapascan-0 peut être utilisé directement, mais a été plutôt conçu comme un support pour le niveau Athapascan-1.

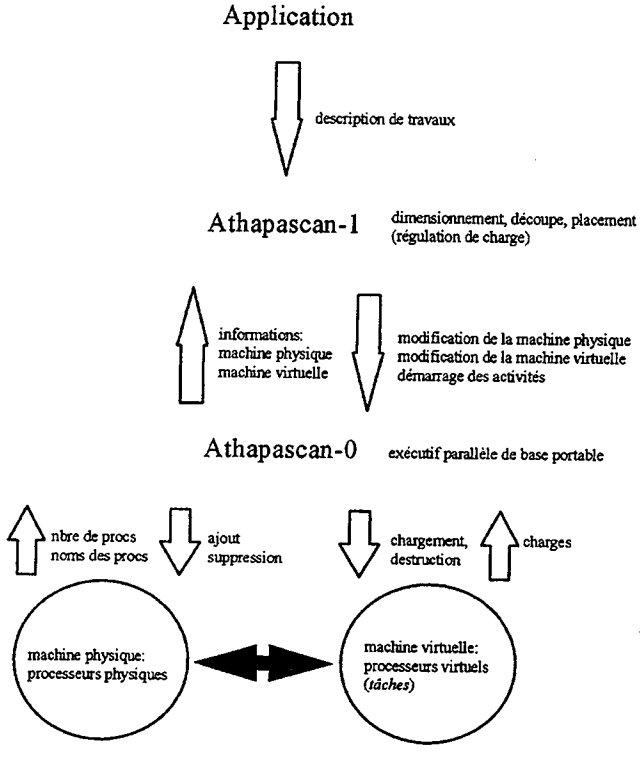

FIG. 2.5 - *L'environnement Athapascan.*

Le niveau Athapascan-1 offre, au-dessus des fonctionnalités du niveau Athapascan-0, un parallélisme explicite, dans lequel seuls les travaux sont décrits et non leur placement, doublé d'une régulation de charge. L'utilisateur du niveau Athapascan-1 peut décrire les travaux à réaliser d'une façon

plus générique qu'en listant toutes les activités du niveau Athapascan-0 nécessaires à leur réalisation. Le niveau Athapascan-1 prend en charge cette description et la transforme en un ensemble d'activités Athapascan-0. Pour effectuer cette transformation le niveau Athapascan-1 utilise des algorithmes prédéfinis qui puisent leurs décisions dans les informations (taille de la machine physique, taille de la machine virtuelle, états de charge) rapportées par le niveau Athapascan-0. Le niveau Athapascan-1 fait l'objet de recherches spécifiques, en relation étroite avec le niveau Athapascan-0, mais poursuivant des objectifs distincts.

2.4.3 Portabilité et efficacité : les poly-algorithmes

La notion de portabilité définit la réutilisabilité d'un code en cas de changement de l'environnement d'exécution. Les changements possibles sont par exemple :

- ajout ou suppression de mémoire
- changement de vitesse du processeur, du réseau
- ajout ou suppression de processeurs
- changement d'architecture de processeurs (par ex. de Power en Alpha)
- changement de topologie réseau
- changement de système d'exploitation
- changement de fonctionnalités (horloge globale, diffusion matérielle, etc...)

En cas de changement de l'environnement d'exécution, il est nécessaire d'adapter plus ou moins profondément l'application. Une *échelle de difficulté de portabilité* peut être définie selon la modification du code qu'il est nécessaire de réaliser :

- aucune : le code s'exécute encore,
- triviale : simple recompilation,
- facile : quelques modifications à faire,
- compliquée : réécrire certains modules,
- très compliquée : réécrire l'application à la base,
- « impossible » : il faut changer l'application ; elle n'est plus possible sur le système cible.

Une *échelle de qualité de portabilité* doit aussi être définie. La notion de qualité peut varier en fonction de ce qui est important : cela peut par exemple être le temps de réponse, le débit, la fonctionnalité... Une telle échelle peut par exemple être :

- la qualité du code porté est médiocre,
- la qualité du code porté est comparable,
- la qualité du code porté est proportionnelle au changement effectué,
- la qualité est exceptionnelle (supérieure au changement effectué).

Nous prendrons comme définition commune d'un **portage efficace** « un portage dont la qualité est proportionnelle au changement effectué ». Dans le contexte qui nous intéresse ici, cela implique que si l'on passe d'une machine parallèle à une autre offrant une puissance double, l'application portée devra avoir un temps d'exécution réduit de moitié. Il est bien connu que cette propriété n'est pas facile à réaliser.

Athapascan-0 assure à ses applications un certain degré de portabilité puisqu'il définit une couche portable offrant une machine virtuelle. Athapascan-0 essaie de garantir une efficacité optimale, quelque soit la machine, pour les fonctionnalités élémentaires (comme le démarrage d'une action à distance). Cependant, Athapascan-0 ne peut pas garantir l'efficacité de l'application portée. Par exemple, si l'application initiale est développée pour 1000 processeurs, il y aura sûrement un surcoût important lié à la fourniture de 1000 processeurs virtuels sur une machine qui n'a que 10 processeurs physiques. Le

niveau Athapascan-1 et l'application doivent donc s'entendre pour prendre en charge cette portabilité efficace. Le niveau Athapascan-1 ne peut pas, à lui seul, garantir l'efficacité, puisque selon les caractéristiques de la machine et du travail à faire, tel ou tel algorithme, pour le même résultat, sera plus performant. Le niveau Athapascan-1 ne peut pas inventer un nouvel algorithme, ou même seulement choisir arbitrairement entre deux algorithmes, il peut seulement effectuer le découpage et le placement des activités d'un algorithme. Nous pensons donc que l'application doit s'adapter à l'environnement ambiant de façon à être efficacement portable. L'application doit choisir un « bon » algorithme en fonction des informations (nombre de processeurs, états de charge, ratio calcul / communication...) rapportées par Athapascan-0.

Ce type de programmation nous entraîne sur la notion de **poly-algorithmes** [Snir 1992], c'est à dire qu'un travail est décrit non pas par un seul algorithme, mais par un ensemble d'algorithmes possibles et une règle de choix qui doit déterminer quel algorithme utiliser dans un cas particulier, en fonction des paramètres du travail (taille des données à traiter, etc..), des paramètres des machines physiques et virtuelles (nombre de processeurs, granularité..) et de leur état (charges..) [Vermeerbergen 1994, Roch et al. 1994]. Cette notion est illustrée ci-dessous :

```
algorithme A(entrée : travail, sortie : résultat)
début
règle(paramètres du travail, paramètres des machines physiques et virtuelles) -> algo
choix selon (algo)
cas a : algorithme a(travail, résultat)
cas b : algorithme b(travail, résultat)
cas c : algorithme c(travail, résultat)
...
cas z : algorithme séquentiel(travail, résultat)
fin
```

FIG. 2.6 - *Un poly-algorithme.*

La règle permet le choix de l'algorithme à employer dans des conditions particulières. Cet algorithme choisi peut se décomposer en plusieurs sous-algorithmes parallèles. Le choix de la découpe (nombre de sous-algorithmes, placement de ceux-ci) est du ressort de l'interaction entre l'application et le niveau Athapascan-1. En guise d'exemple, l'algorithme *a* pourrait être écrit comme suit :

```
algorithme a(entrée : travail, sortie : résultat)
début
décider d'une découpe en collaboration avec la couche de régulation de charge Athapascan-1
découper le travail en un ensemble de sous-travaux
calculer tous les sous-travaux en parallèle selon leurs « sous-algorithmes »
rassembler les résultats en un résultat final
fin
```

FIG. 2.7 - *Détail d'un algorithme parallèle.*

Pour aider le programmeur dans l'expression du parallélisme, le niveau Athapascan-1 doit fournir des opérateurs adéquats pour effectuer la découpe des paramètres et le rassemblement des résultats. Il doit aussi permettre la « collaboration » des sous-algorithmes parallèles. Enfin, il doit permettre le choix d'un algorithme et d'une découpe particulière, en informant l'application des caractéristiques de la machine cible. Ces informations sont en partie extraites du support d'exécution de base, Athapascan-0.

Le programmeur exprime donc un ensemble d'algorithmes pour chaque problème, censé être suffisant pour autoriser une portabilité efficace sur une large gamme de machines et de conditions de

fonctionnement. Il exprime le parallélisme, mais il n'explicite pas le placement des activités. C'est la couche de régulation de charge, à définir par le niveau Athapascan-1, qui effectuera ce placement. D'autre part, l'approche prise est une approche procédurale : chaque algorithme est une procédure. Les avantages de cette approche sont bien connus. Elle permet :

- une décomposition claire d'un problème en sous-problèmes ; et à l'inverse la possibilité de composer simplement un algorithme à partir de briques de base,
- l'identification précise du dialogue entre l'appelant et l'appelé, limitant les risques d'erreurs,
- une désignation contextuelle des variables à l'intérieur d'un espace de nommage bien défini.

En conséquence, la réutilisation de code sera grandement facilitée par cette approche. Nous souhaitons que le programmeur d'applications puisse facilement combiner des briques de base, en profitant à la fois d'un bon grain de parallélisme et d'une bonne régulation de charge, sans pour autant devoir les expliciter.

Notons que chaque sous-algorithme issu d'une décomposition parallèle peut être lui-même un poly-algorithme. Il y a donc une découpe parallèle des poly-algorithmes, qui se poursuit jusqu'à ce qu'un algorithme séquentiel soit choisi pour chaque bout de travail restant. Avec Athapascan, chaque sous-algorithme est exécuté par un appel de procédure à distance qui est placé par le niveau Athapascan-1 de régulation de charge. Le mécanisme de base que doit pouvoir réaliser Athapascan-0 est donc l'appel à distance d'une **procédure parallèle** : l'appel en parallèle de plusieurs procédures sur des sites éventuellement différents, avec des paramètres et des résultats différents.

Cet appel de procédure parallèle est en fait implanté avec un mécanisme de RPC asynchrone. Une requête est envoyée pour chaque procédure devant être appelée à distance. Le processus client qui émet les requêtes n'attend pas immédiatement leurs résultats : d'où la mention de RPC *asynchrone*. Une fois toutes les requêtes émises, il attend les réponses une à une. Sur les sites serveurs, un nouveau processus est créé lors de la réception de chaque requête. Ce processus déroule le sous-algorithme correspondant, et fini par transmettre son résultat dans un message de réponse avant de se terminer. Nous expliciterons plus en détail ce mécanisme dans le chapitre 4.

2.4.4 Interaction avec les autres axes d'APACHE

Nous avons indiqué les grandes lignes du support d'exécution que nous voulons réaliser, en corrélation forte avec le niveau Athapascan-1 de régulation de charge. Nous allons maintenant décrire les interactions entre l'exécutif et les autres axes du projet APACHE : déverminage, prise de trace, observation, rétroaction, applications.

2.4.4.1 Déverminage et réexécution déterministe

L'indéterminisme d'une application parallèle grève de façon très importante les possibilités de déverminage de l'application. Une solution possible pour assurer un déverminage dans des conditions idéales est de réexécuter l'application dans la même « séquence » que lorsque l'erreur initiale c'est produite. Comme l'erreur peut s'être produite lors d'une très longue exécution, difficilement rejouable à cause de sa longueur ou bien parce que l'erreur est très indéterministe, l'idéal serait de pouvoir enregistrer la « séquence » de toute exécution, de façon à pouvoir la rejouer sous un déverminateur si une erreur se produit. La question qui se pose alors concerne le coût de cet enregistrement automatique et donc le choix de ce que l'on enregistre. Pour être viable, une telle solution ne doit pas consommer une quantité trop importante de ressources. Dans le cadre de son DEA et de sa thèse, Alain Fagot a démontré ([Fagot & de Kergommeaux 1996, Fagot & de Kergommeaux 1995*a*, Fagot & de Kergommeaux 1995*b*, de Kergommeaux & Fagot 1995, Fagot & de Kergommeaux 1994]) qu'une

solution possible à ce problème est de n'enregistrer que l'ordre des événements indéterministes et les valeurs non reproductibles de ces événements. Le surcoût de cette solution reste négligeable. La base de la solution proposée est principalement d'enregistrer uniquement l'ordre des prises en compte des requêtes ou réponses de RPC. En rejouant ces événements dans le même ordre, on garanti une réexécution déterministe. Cette solution suppose que l'ordonnancement des activités à l'intérieur d'une tâche soit déjà déterministe. C'est le cas dans l'implantation 0a d'Athapascan grâce à la simplicité de son modèle d'exécution, comme nous le verrons plus loin.

2.4.4.2 Prise de trace

Pour évaluer les performances d'une application [Teodorescu & de Kergommeaux 1995, Arrouye 1995], il est nécessaire de prendre des traces de l'exécution de l'application. Les problèmes de cette prise de trace sont de deux sortes : premièrement, le coût de la prise de trace, qui peut être prohibitif ou bien entraîner un comportement différent de l'application tracée et de l'application non tracée, dû à des synchronisations perturbées par les opérations de traçage. Deuxièmement, la nécessité de modifier le support d'exécution pour tracer le comportement de celui-ci et à travers lui le comportement de l'application. Bien que le premier problème nécessite une étude spécifique dans le cadre de la prise de traces, le second peut être allégé par l'utilisation d'un mécanisme de rappels (*callbacks*), qui permet d'insérer des points de trace facilement et relativement indépendamment du fonctionnement du support d'exécution. Les concepteurs d'outils d'évaluation de performances ont donc défini dans l'exécutif Athapascan-0 un petit ensemble de points de traçage sur lesquels ils ont accroché des rappels vers le code de prise de trace, chaque rappel spécifiant le type d'événement tracé et son identification (par ex : une procédure a été appelée à distance, il s'agit de la procédure *xxx*). La définition de rappels permet de séparer le code de prise de trace du code de l'exécutif. Cette approche permet aussi d'implanter plusieurs traces (par ex. celle pour la réexécution déterministe, celle pour l'évaluation de performances, celle pour le déverminage de l'exécutif), sans modification excessive de l'exécutif, les rappels s'empilant les uns sur les autres de façon automatique. Notons que si cette réalisation est particulièrement simple à mettre en œuvre, ceci est encore dû à la simplicité du modèle d'exécution d'Athapascan- 0a, qui limite la concurrence.

2.4.4.3 Instrumentation

L'instrumentation est nécessaire pour obtenir des informations quantitatives sur le fonctionnement de l'application, in-vivo (sans passer par une analyse de traces post-mortem). Cette instrumentation sert principalement à la prise de décision concernant la découpe et la régulation de charge effectuées au niveau Athapascan-1. Actuellement, l'instrumentation proposée concerne :

- le nombre de machines physiques disponibles et
- une certaine notion de la charge instantanée « Athapascan-0 » d'un processeur virtuel. Cette charge instantanée est définie comme deux valeurs,
  - le nombre total d'activités en cours sur le processeur virtuel et
  - le nombre d'activités qui sont prêtes à tourner.

Cette charge ne tient pas compte des travaux annexes effectués par le processeur, par exemple pour le compte d'un autre utilisateur. L'instrumentation fournie par l'exécutif doit être complétée par des informations de plus bas niveau (charge Unix (*rusage*), taux de pagination, ...) ou de plus haut niveau, gérées par le régulateur lui-même. Le noyau Athapascan-0 n'a pas l'ambition de fournir un indice de charge général, il laisse cette responsabilité à la couche de régulation de charge [Castaneda-Retiz & Plateau 1996], mais propose simplement d'exporter quelques informations qui peuvent être utiles et qui sont de son ressort.

Notons que certaines de ces informations peuvent être établies par le régulateur lui-même, par construction. Ainsi en est-il du nombre total d'activités ; puisque le régulateur les place, il n'a pas de mal à les compter. Dans le même ordre d'idées, le nombre de processeurs virtuels n'est pas rendu disponible par l'exécutif, puisque l'application doit elle même gérer ses processeurs virtuels. Elle peut donc aussi les compter.

2.4.4.4 Rétroaction d'ordonnancement

Toutes les informations d'instrumentation servent à établir les décisions de régulation. Athapascan-0 fournit comme rétroaction possible de la part du régulateur la possibilité :

- d'agir sur les machines physiques et virtuelles (ajout et suppression de processeurs et de tâches),
- de démarrer une activité à une endroit donné et
- de suspendre et de réveiller certaines activités à l'aide de mécanismes de synchronisation.

2.4.4.5 Applications

La facilité de développement d'applications parallèles irrégulières est un des buts du projet APACHE. La réalisation de ce but ne peut être évalué qu'à l'aune de l'implantation et la mesure d'applications réelles.

Le niveau 1 d'Athapascan est particulièrement novateur et demande donc un investissement certain de réalisation. Les premières applications du projet APACHE ont donc été écrites directement au-dessus du niveau 0. Ces applications se répartissent en une collection d'applications triviales de démonstration et de test, une application simple parallélisant un programme en Athapascan-0 de lancé de rayons, et enfin des applications « grandeur nature » :

- un compilateur parallèle PROLOG [Briat et al. 1996, Briat et al. 1995*a*, Briat et al. 1995*b*],
- une bibliothèque pour le calcul formel réalisée en C++, PAC++ [Gautier 1996, Gautier & Roch 1994, Gautier et al. 1994] et
- une application de dynamique moléculaire [Bernard et al. 1996*b*, Bernard et al. 1996*a*, Bernard & Trystram 1996].

Ces trois dernières applications résultent elles-mêmes des axes de recherche du projet APACHE. Nous ne les décrivons pas ici, nous laissons le lecteur se reporter aux documents publiés par leurs créateurs. Une évaluation d'efficacité relative de PAC++ au dessus d'Athapascan-0 et au dessus des communications natives est donnée en 4.7.6.

2.5 Conclusion

Nous avons présenté les architectures de machines parallèles les plus courantes et retenu un modèle particulier de machine parallèle. Nous avons reconnu un paramètre fondamental : le grain de parallélisme. Cette variabilité de la machine parallèle est à notre avis l'un des points clés dont il faut tenir compte pour réaliser des applications parallèles portables. Nous avons décrit divers paradigmes de parallélisme et indiqué que quelque soit le paradigme employé, le contrôle de la granularité, nécessaire pour réaliser la portabilité efficacement, doit être effectué. Nous avons introduit les supports d'exécution parallèles comme la base pour l'expression *explicite* du parallélisme. Nous avons abordé les deux concepts de base de ces exécutifs : les processus et les canaux de communication. Différentes approches de ces concepts sont possible, selon la dynamicité voulue, le coût de la mise en œuvre et le contrôle d'exécution possible. Différents modèles d'interaction entre processus ont été abordés : la

communication par échange de messages entre processus préexistants, la transmission d'information lors des créations et terminaisons de processus et l'accès unilatéral aux mémoires distantes. Notons que le choix d'un modèle d'interaction particulier et des opérateurs associés dépend du paradigme de parallélisme souhaité et du champ d'application ciblé.

Nous avons vu que les applications parallèles *irrégulières* nécessitent un support d'exécution spécialisé sur les machines parallèles modernes. Ce support doit prendre en compte l'aspect dynamique, concurrent et à grain indéterminé de ces applications. Nous nous sommes donné comme objectif la réalisation d'un tel support par la combinaison d'un mécanisme de processus léger avec une bibliothèque portable pour le parallélisme. Nous avons décrit l'approche du projet APACHE : la mise en œuvre de deux couches exécutives, l'une prenant en charge les mécanismes de base d'un parallélisme explicite, l'autre effectuant automatiquement la régulation de charge. Nous avons introduit le concept de poly-algorithme pour l'expression d'applications efficacement portables. L'appel de procédures parallèles est le mécanisme de base de ce concept ; le modèle d'interaction que nous avons choisi dans le cadre du projet APACHE est donc tout naturellement un modèle client-serveur dans un paradigme de parallélisme de contrôle.

Le projet APACHE se décompose en plusieurs axes de recherche : l'étude d'un support exécutif de base, la régulation de charge, la mise au point des performances, le déverminage parallèle, la programmation d'applications. Chaque axe a sa propre thématique. L'axe concernant l'exécutif de base a une thématique restreinte que nous allons développer dans la suite. En particulier, nous avons volontairement exclu de cette thématique les aspects fondamentaux des autres axes de recherche. C'est pourquoi nous ne parlerons que très peu dans la suite de choix de découpe, de régulation de charge et d'efficacité d'applications. Nous renvoyons le lecteur aux différents documents produits par le projet APACHE, que nous avons cités précédemment. Nous avons simplement présenté ici l'interaction entre ces différents axes et l'exécutif de base.

Nous allons maintenant examiner deux mécanismes utiles pour la réalisation d'un support exécutif qui réalise l'expression que nous avons décrite.

Chapitre 3

Mécanismes utilisables pour la réalisation d'un support exécutif pour applications parallèles irrégulières

Nous allons maintenant nous intéresser plus particulièrement aux mécanismes utiles pour la réalisation d'un support exécutif pour applications parallèles irrégulières. Nous ne présenterons pas ici les mécanismes qui permettent de créer des processeurs virtuels ; ces mécanismes s'apparentent à la création d'un processus lourd à distance et sont relativement bien connus. Par contre, nous examinerons un mécanisme d'apparition relativement récente - sous son expression actuelle - la multiprogrammation légère. Nous détaillerons ensuite une bibliothèque de communication particulière, PVM, qui est intéressante puisqu'elle donne un bon exemple de la réalisation des communications dans une machine parallèle, au-dessus d'un système d'exploitation classique.

3.1 La multiprogrammation légère

La multiprogrammation légère est, comme nous l'avons vu, potentiellement très utile dans le cadre d'applications parallèles. Nous allons maintenant décrire les buts principaux de cette technique, le mécanisme de base, les implantations possibles, les problèmes posés et enfin nous passerons en revue quelques implantations, notamment celles que nous avons utilisées pour Athapascan-0.

3.1.1 Origine

La notion de flot d'exécution existe quasiment depuis l'origine de l'informatique. Dijkstra en fait par exemple référence en 1965 [Dijkstra 1965]. Une implantation prototype de flots d'exécution a été faite dans les années 1970 dans Multics. Par la suite, Unix a associé le flot d'exécution à un espace d'adressage, créant la notion de processus lourd. Les processus légers tels que nous les connaissons datent des travaux effectués sur MACH et les C- threads [Accetta et al. 1986], qui furent influencés par la conception de la machine DEC Firefly [Bershad et al. 1988] et du langage MODULA-2+ [Rovner et al. 1985], eux-mêmes influencés par le langage MESA [Lampson & Redell 1980] des années 1980. Une introduction sur la multiprogrammation aussi bien lourde que légère peut être trouvée dans [Tanenbaum 1992].

3.1.2 Buts principaux

Les buts principaux sont multiples. Le fait de pouvoir utiliser plusieurs activités, chacune avec son contexte, gérées d'une façon « légère » permet d'*accroître le parallélisme d'une application*. Il est ainsi facile de décrire une application qui est intrinsèquement parallèle, comme par exemple un *pipeline* de tâches.

Cela permet aussi une certaine *indépendance vis à vis de la machine physique*, une application multiprogrammée pouvant utiliser un nombre indéterminé de processeurs en même temps, en répartissant les différentes activités (**fils d'exécution** ou *threads*) sur les différents processeurs. C'est pourquoi c'est une technique importante pour l'*utilisation efficace des multiprocesseurs à mémoire partagée* (SMP), qui a été particulièrement développée dans le cadre des systèmes d'exploitation dédiés à ces machines.

D'autre part *la réactivité d'une application est améliorée*, et cela au-delà de ce qu'il est possible de faire avec des gestionnaires d'interruption. Il est en effet relativement facile de démarrer un fil d'exécution pour chaque nouvel événement reçu, qui va prendre en charge cet événement sans bloquer l'application ; il est plus difficile d'écrire des gestionnaires d'interruption réentrants[1] , capables d'effectuer de « longues » opérations.

De plus la possibilité d'obtenir *une exécution concurrente de plusieurs tâches indépendantes*, par la technique du *partage de temps* entre les différents fils d'exécution, est utile dans certains types de problèmes, en particulier du genre *branch and bound*, et permet dans certains cas des accélérations super-linéaires.

La *simplification d'écriture d'applications* qui en résulte n'est pas négligeable : au lieu de concevoir un automate séquentiel qui épuise les événements les uns après les autres, dont le fonctionnement est bien souvent difficile à prouver, ou bien une série de gestionnaires d'interruptions asynchrones, pour lesquels il est difficile d'établir la réentrance et la synchronisation, les fils d'exécution, indépendants, permettent d'écrire un traitement synchrone des différents événements, en toute indépendance les uns des autres.

Accessoirement, la multiprogrammation légère permet d'*éviter les interblocages* qui se produisent par exemple lors de l'interaction de plusieurs serveurs réentrants.

Enfin, il est possible d'obtenir un *recouvrement calcul / communication* [Boothe & Ranade 1992, Felton & McNamee 1992*b*] automatiquement : un processus peut avoir plusieurs opérations d'entrées-sorties pendantes et être toujours constitué de séquences bien déterminées. Sans la multiprogrammation il est nécessaire de mettre en œuvre des techniques d'entrée-sortie asynchrone, qui sont difficiles à manipuler.

Le dernier but est d'*effectuer des communications à moindre coût*, dans une application constituée de plusieurs activités, puisque l'utilisation de la mémoire partagée est généralement moins coûteuse que la communication inter-processus (IPC).

Notons que le recouvrement calcul / communication, qui semble à priori être très intéressant pour masquer les coûts de communication dans les programmes parallèles, est rapporté dans la littérature comme étant accessoire vis à vis de la facilité d'écriture des applications [Fahringer et al. 1995, Haines & Böhm 1993*a*, Haines & Böhm 1993*b*] : un programme codé manuellement sous la forme d'un automate séquentiel ou d'une série de gestionnaires d'interruptions asynchrones est forcément plus rapide qu'un programme à base de processus légers. Cependant il peut être beaucoup plus difficile à concevoir et à mettre au point.

1. Ce terme est défini un peu plus loin

3.1.3 Description générale

Le mécanisme principal liée à la multiprogrammation légère est le changement de contexte léger. Dans une **multiprogrammation lourde**, chaque activité possède un fil d'exécution, un contexte d'exécution « interne » (pile, registres...), un espace d'adressage et de protection et un contexte d'exécution « externe » (fichiers ouverts, positions et verrous sur ces fichiers, masque de signaux...; ce contexte est partagé avec le reste du système). Cet ensemble forme ce que l'on appelle couramment un **processus**. Dans la multiprogrammation lourde, les processus peuvent occuper à tour de rôle le processeur. L'action de commutation d'un processus vers un autre consiste à sauvegarder les caractéristiques du processus en cours, élire le prochain processus et restaurer les caractéristiques de ce nouveau processus. Cette action de commutation peut intervenir à des points d'exécution particulier des processus, auquel cas on parle de **multiprogrammation coopérative**, ou bien n'importe quand, auquel cas on parle de **multiprogrammation préemptive**. Le problème de la multiprogrammation lourde est que, puisque l'on doit modifier l'espace d'adressage et le contexte d'exécution externe, une intervention du système d'exploitation est nécessaire pour toute opération de gestion des processus. Outre le coût d'appel au système, cette intervention demande une quantité de travail importante puisqu'il faut changer d'espace d'adressage et de protection.

Dans la **multiprogrammation légère** au contraire, plusieurs activités, **fils d'exécution** doublés d'un contexte d'exécution « interne » propre, se partagent un unique espace d'adressage et de protection et les principales caractéristiques d'un contexte d'exécution « externe ». Lorsque l'espace d'adressage et le contexte externe sont en place, l'action de commutation de fils d'exécution ne nécessite que la commutation du contexte d'exécution interne et donc permet d'éviter l'appel au système et le coût du changement d'espace d'adressage et de protection. C'est pourquoi le délai de commutation est beaucoup plus petit, typiquement 10 à 1000 fois. De même la création et la destruction d'un processus léger ne demandent que l'allocation et la libération d'un contexte interne.

Le mécanisme de commutation doit sauvegarder le contexte interne du fil d'exécution courant et restaurer le contexte interne de son successeur. Cette opération dépend du contexte interne qui est principalement défini par le compilateur; elle peut être écrite en assembleur, d'une façon non portable, ou bien à l'aide des primitives de saut non-local setjmp() et longjmp() qui sont relativement standards sous Unix (norme POSIX), mais bien souvent incomplètes (pas de sauvegarde de l'état du coprocesseur mathématique, par exemple) et moins performantes. La multiprogrammation légère peut aussi être coopérative ou préemptive.

La multiprogrammation légère et la multiprogrammation lourde ne sont pas exclusives; on peut très bien avoir plusieurs processus se partageant un processeur, certains d'entre eux étant multiprogrammés de façon légère (en anglais on parle de *multithreading*). La multiprogrammation légère doit collaborer avec la multiprogrammation lourde pour négocier la part de puissance du processeur physique attribuée à un processus. Deux solutions extrémistes sont possibles. Un processeur physique peut supporter:

- un seul processus lourd contenant n processus légers: toute la puissance est gérée par la multiprogrammation légère,
- n processeurs lourds contenant chacun un seul processus léger: toute la puissance est gérée par la multiprogrammation lourde.

Une solution mitigée est de supporter n processus légers dans m processus lourds; cependant il faut alors contrôler le nombre de processus légers autorisés à tourner pour un processus lourd, sans introduire pour autant d'interblocage. Nous y reviendrons dans la suite.

3.1.4 Mode d'emploi de la multiprogrammation légère

L'interface de programmation pour la multiprogrammation légère ne diffère pas fondamentalement de celle de la multiprogrammation lourde. Les opérateurs qui permettent d'agir sur les fils d'exécution sont des opérateurs d'existence, d'ordonnancement et de synchronisation. Une application multiprogrammée possède un point d'entrée initial, main(), qui est transformé d'une façon ou d'une autre en un fil d'exécution relativement ordinaire. Il est alors possible de créer de nouveaux fils d'exécution, ou bien de les détruire. Le démarrage d'un nouveau fil d'exécution se fait en créant un contexte interne arbitraire, de telle sorte que le fil d'exécution exécute une fonction donnée. Lorsque le fil d'exécution termine (sort de) la fonction, il est automatiquement détruit. L'identification d'un fil d'exécution se fait généralement en donnant une référence sur son contexte interne. Les opérateurs nécessaires à l'ordonnancement concernent la suspension et la réactivation de fils d'exécution, c'est à dire la gestion de ce qui est en général réalisé comme une (ou plusieurs) liste(s) de fils d'exécution prêts. Bien souvent on introduit un ordonnancement à base de priorités, permettant de classer l'urgence des différentes activités. Enfin les opérateurs de synchronisation gèrent des sémaphores, verrous, variables de condition ou bien moniteurs [Hoare 1974] pour assurer des accès cohérents au données partagées par les différents fils d'exécution.

Le modèle de vie des fils d'exécution peut être libre (chacun pouvant créer de nouveaux fils et tuer n'importe lequel, y compris soi-même), ou bien plus contraint, suivant le modèle *fork-join* dans lequel le fil d'exécution parent peut, après avoir effectué quelques traitements, se mettre en attente de la terminaison d'un de ses fils et lors de cette synchronisation, récupérer quelques résultats. Un fonctionnement mixte est aussi tout à fait possible.

La plupart des noyaux de processus légers associent une zone mémoire privée à chaque fil d'exécution. Cela permet une extension simple de son contexte, utilisable par le programmeur.

3.1.5 Implantations possibles et problèmes posés

Dans cette section nous allons brièvement introduire les problèmes posés lors de l'utilisation de la multiprogrammation légère. Ces problèmes sont de deux types : principalement ceux liés à la concurrence introduite dans l'application, mais aussi l'interaction avec le système d'exploitation, la gestion des deux niveaux de multiprogrammation, de la préemption, de l'ordonnancement, des signaux, des piles et des contextes.

3.1.5.1 Problèmes liés à la concurrence

L'introduction de la concurrence dans une application peut aboutir à des **conflits d'accès aux données** (*data races*) : quand un processus léger écrit une donnée alors qu'un autre tente de la lire. Il faut alors verrouiller l'accès aux données pour garantir une exécution cohérente. Ce verrouillage peut introduire des **interblocages** (*deadlocks*) si par exemple deux processus légers ont chacun verrouillé une donnée et essayent chacun d'accéder la donnée de l'autre. De même il peut se produire une **famine** (*starvation*) si un processus léger n'arrive pas à accéder aux données qui sont toujours verrouillées par un autre. Un certain nombre de problèmes de performances peuvent aussi apparaître :

- déséquilibre du nombre de processus légers entre chaque processeurs,
- coût des changements de contexte,
- coût de la synchronisation entre les processus légers,
- coût de l'accès aux données,
- contention sur les verrous qui les protègent...

Nous ne nous attaquerons pas ici aux problèmes liés aux conflits d'accès ou aux interblocages, dont les solutions dans le cadre de la programmation concurrente sont maintenant bien connues [Helm-Bold & McDowell 1994, Doeppner 1993, Jones 1991].

3.1.5.2 Interaction avec le système d'exploitation : deux niveaux de multiprogrammation

Ce qui a été dit plus haut à propos des multiples activités d'une application multiprogrammée, commutées sans l'aide du système d'exploitation, n'est que partiellement vrai. En effet, si le système ne connaît pas les multiples fils d'exécutions, il ne peut pas bloquer sélectivement l'un d'eux sans bloquer toute l'application. Cela se produit par exemple si un fil d'exécution effectue un appel système qui se révèle être bloquant. Le système d'exploitation ne connaissant pas les autres fils d'exécution de l'application, il ne peut pas commuter vers un autre fil prêt. L'application en entier est donc bloquée. A l'inverse, si tous les fils d'exécution étaient connus du système, cela aurait deux effets pervers au moins. Premièrement la nécessité pour le système de réserver des structures de données pour gérer chaque fil d'exécution, ce qui entraînerait un écroulement du système ou bien une limitation arbitrairement basse du nombre de fils d'exécution simultanément existants. Deuxièmement la nécessité d'informer le système de la plupart des événements intervenant sur les fils d'exécution (création, destruction, changement d'état,...) et donc un surcoût certain dans la gestion des fils d'exécution.

Une approche intermédiaire consiste à définir deux niveaux de fils d'exécution. Un premier niveau de fils d'exécution sera connu par le système, qui recevra toutes les informations nécessaires sur ces fils d'exécution et sera capable d'agir sur leur état en fonction de ce qu'ils font dans le noyau du système d'exploitation. Ces fils d'exécution seront dits « **fils d'exécution noyaux** » (*kernel threads*). Le système d'exécution sera donc capable de commuter un fil d'exécution noyau qui se bloque dans le noyau, au profit d'un autre fil d'exécution noyau prêt. Ce niveau correspondra à des activités de poids intermédiaire, plus légers que les processus lourds (car il n'est pas nécessaire de commuter les espaces d'adressage et de protection), mais plus lourds que ceux du deuxième niveau. Les fils d'exécution du deuxième niveau sont dits « **fils d'exécution utilisateurs** » (*user-level threads*), car le noyau n'est pas au courant de leur existence, et ils sont entièrement gérés dans une couche de support d'exécution liée au code de l'application. Les fils d'exécution utilisateurs sont associés aux fils d'exécution noyaux dans un rapport N vers M plus ou moins lâche ($N \geq M$). Par exemple, un fil d'exécution utilisateur peut être lié (*bound*) à un fil d'exécution noyau. Leurs destins sont donc communs, si l'un se bloque, l'autre aussi. A l'inverse dix fils d'exécution utilisateurs peuvent être associés à deux fils d'exécution noyau. Les fils d'exécution utilisateur sont alors liés de façon temporaire aux fils d'exécution noyaux. Si l'un des dix fils d'exécution utilisateur, représenté par l'un des deux fils d'exécution noyau, se bloque dans le noyau, l'autre fil d'exécution noyau pourra toujours représenter les neufs autres fils d'exécution utilisateur. L'application n'est donc pas bloquée dans son ensemble. Le nombre de fils d'exécution du niveau utilisateur n'est pas limité par les ressources du noyau. A l'inverse le nombre de fil d'exécution noyau représente le degré de parallélisme réel souhaité par l'application et chacun étant une entité différente, l'application peut avoir une meilleure souplesse, par exemple en leur définissant des tâches bien différenciées qui nécessitent des caractéristiques différentes.

Ce système à deux niveaux est donc souple et performant. Il est utilisé dans Solaris [Powell et al. 1991, Stein & Shah 1992, Eykholt et al. 1992] et a aussi été décrit par [Anderson et al. 1992, Barton-Davis et al. 1992]. Un problème cependant réside dans le choix du nombre de fils d'exécution noyau qui doivent être associés à une application. Si ce choix est trop petit, tous les fils d'exécution noyau peuvent être bloqués, ce qui bloque l'application même s'il reste des fils d'exécution utilisateurs prêts. De plus les fils d'exécution utilisateurs peuvent être interbloqués, si l'application est bloquée arbitrairement alors que le fil d'exécution qui débloquerait l'application est prêt mais ne peut pas s'exécuter puisqu'il n'y a plus de fil d'exécution noyau libre. Pour éviter ce problème les ingénieurs de SunSoft créent de nouveaux fils d'exécution noyaux chaque fois qu'il en manque, puis les réclament

s'ils non pas été utilisés pendant un certain laps de temps [Stein & Shah 1992]. Le parallélisme réel de l'application est donc adaptatif. Voir aussi à ce sujet [Inohara & Masuda 1994]

3.1.5.3 Préemption et interruption

Notons la différence qu'il y a entre préemption et interruption. L'**interruption**, c'est le fait d'interrompre une activité, d'effectuer un travail annexe, puis de reprendre l'exécution de l'activité interrompue. Ce mécanisme est matériel, et généralement répercuté par le système d'exploitation au niveau du programmeur d'application. La question importante, c'est ce que peut faire le traitant d'interruption. Il peut effectuer une action complètement indépendante du traitement interrompu ; cela ne pose pas de problème. Par contre, s'il doit collaborer avec l'action interrompue, par exemple parce qu'il va lire sur disque les blocs demandés par le traitement principal, alors il se pose un problème de synchronisation entre les deux activités. L'interruption doit être masquée durant certaines périodes critiques, sinon les données partagées seront accédées de façon incorrecte. Ce problème général conduit à limiter les actions possibles dans le traitant d'interruption, à un petit nombre d'opérations clairement identifiées.

La **préemption**, c'est le fait non seulement d'interrompre le processus en cours, mais en plus de commuter vers un autre processus (en le créant éventuellement). Le premier processus reprendra son exécution plus tard, sous le contrôle de l'auto-ordonnanceur des processus. Nous désignons ici sous le terme d'**auto- ordonnanceur** une routine suivant une politique déterminée pour ordonnancer les processus, sans intervention extérieure. L'utilisation de la préemption implique un contrôle encore plus serré des données partagées et des actions effectuées, puisque l'intérêt est de ne pas limiter les processus dans les opérations qu'ils peuvent faire, contrairement au cas d'un traitant d'interruption. Dans un mode préemptif, la commutation peut avoir lieu à n'importe quel moment, par exemple sur une entrée-sortie ou un signal d'horloge. L'avantage de cette organisation est de permettre un très bonne réactivité aux événements asynchrones ou un partage de temps entre les différents fils d'exécution. Le problème est qu'il est nécessaire de verrouiller toute donnée partagée et d'assurer la réentrance des routines utilisées. Un code exécuté plusieurs fois en concurrence est un **code réentrant** s'il produit dans tous les cas un résultat correct sans nécessiter aucune synchronisation ou exclusion externe. Cela aboutit à deux principes :

- l'utilisation systématique d'un contexte différent (pile par exemple) pour chaque exécution.
- et/ou l'exclusion mutuelle (désactivation des interruptions, utilisation de sémaphores ou de moniteurs) sur les portions de code qui ne sont pas réentrantes (voire sur le code en entier).

Outre le fait que les verrouillages peuvent entraîner des interblocages, la réentrance d'une bibliothèque de fonctions quelconque est difficile à assurer. C'est pourquoi l'exécution préemptive de processus légers et difficile à maîtriser et nécessite bien souvent des connaissances approfondies en système d'exploitation et en gestion de la concurrence.

A l'inverse, les fils d'exécution peuvent être comme des **coroutines** [Knuth 1973], c'est à dire s'exécuter jusqu'à une instruction de commutation vers un autre fil d'exécution. Souvent l'instruction précise au profit de quel fil d'exécution la commutation sera faite. L'avantage de cette approche sur un monoprocesseur (ou si l'application n'utilise qu'un seul fil d'exécution noyau) est que, lorsqu'un fil d'exécution tourne, on sait qu'il est en exclusion et en conséquence tout verrouillage des données partagées est inutile. Le fait d'avoir le contrôle des commutations permet de contrôler aisément les accès partagés, et donc d'utiliser des fonctions qui ne sont pas réentrantes. Un inconvénient de cette approche est que si le fil d'exécution actif se bloque, par exemple dans un appel système, toute l'application est bloquée. Un autre de ses inconvénients est de ne pas autoriser un parallélisme réel : sur un multiprocesseur à mémoire partagée, les coroutines ne peuvent utiliser qu'un seul processeur à la fois, tout en conservant un fonctionnement correct.

3.1.5.4 Auto-ordonnancement des fils d'exécution

L'auto-ordonnancement de fils d'exécution est généralement assujetti à différents niveaux de **priorité**. C'est alors toujours un fil d'exécution du plus haut niveau de priorité qui tourne. Si plusieurs fils d'exécution sont du même niveau de priorité, un coroutinage, ou bien un partage de temps peuvent être organisés. Dans un mode préemptif, si un fil d'exécution de plus haute priorité est créé ou réactivé, il préempte le fil d'exécution en cours. Un tel mécanisme de priorités est particulièrement utile dans le cadre des systèmes temps réel, qui doivent assurer un temps de réponse minimal à tout événement important. Un problème réside dans l'**inversion de priorité** qui se produit si un fil d'exécution verrouille une donnée, avant un autre fil de plus haute priorité. Le fil de plus haute priorité préempte alors le premier, mais ne peut pas verrouiller la donnée à son tour. S'il tente une attente active pour le verrouillage, un interblocage se produit même. Ce type de problème peut être éliminé en haussant la priorité du fil d'exécution qui a verrouillé une donnée, à la priorité de celui qui tente de verrouiller la donnée jusqu'à ce qu'elle soit déverrouillée.

Il existe trois grandes formes d'auto-ordonnancement des fils d'exécution, qui sont similaires à l'auto-ordonnancement des processus lourds (à la Unix). Ce sont l'ordonnancement FIFO (*first in, first out*), l'ordonnancement en temps partagé et l'ordonnancement temps-réel.

Dans l'auto-ordonnancement FIFO, les processus légers tournent jusqu'à ce qu'ils se terminent ou se bloquent. A leur réveil, ils sont remis en queue de leur file de priorité. Les demandes d'entrée-sortie font tourner les processus à tour de rôle. Si l'ordonnancement est préemptif, un processus léger peut être interrompu par un autre, plus prioritaire. Les priorités sont fixes. L'intérêt de ce mode est sa simplicité de gestion ; en particulier en fonctionnement coopératif. Son inconvénient est que la date d'exécution d'un fil donné ne peut être bornée.

Dans l'auto-ordonnancement en temps partagé, un processus léger tourne au maximum durant un quantum de temps puis est préempté au profit d'un autre. Eventuellement un processus léger peut n'utiliser qu'une partie de son quantum avant de se bloquer. C'est le cas en particulier des processus légers qui sont interactifs. Une technique bien connue pour éviter qu'ils ne soient trop pénalisés vis à vis des processus légers qui ne font que du calcul est de hausser leur priorité en fonction de la partie de quantum qu'ils n'ont pas utilisée. Les priorités sont donc généralement doubles : une composante fixe qui donne la base de la priorité, et une composante variable qui permet de favoriser les processus légers interactifs. L'intérêt de ce mode est de pouvoir assurer une exécution « de front » de plusieurs fils d'exécution. En particulier, le temps partagé garanti une borne supérieure à la date d'exécution d'un fil (le nombre de fils d'exécution prêts x la durée du quantum). L'inconvénient est qu'il nécessite d'être en mode préemptif, et donc de verrouiller les données et de faire attention à la réentrance.

Enfin, l'ordonnancement temps-réel est toujours préemptif et prioritaire. La priorité d'un processus léger est *calculée* en fonction de la date à laquelle celui-ci devra avoir fini son travail. Il est ainsi possible de forcer une exécution telle que les travaux soient toujours faits à temps.

Les deux premières formes d'ordonnancement sont les plus couramment proposées par les noyaux de processus légers dans le cadre du calcul parallèle.

3.1.5.5 Gestion des signaux

Dans un système Unix, les événements extérieurs sont indiqués par des signaux, masqués éventuellement, ou bien traités par différents gestionnaires d'interruption. Il existe différents signaux pour indiquer différentes événements. Une des questions à propos de fils d'exécution concerne les signaux. Doivent-ils être reçus par tous les fils d'exécution d'un processus, un parmi n'importe lequel d'entre eux, ou bien être masqués pour certains ? Doit-il y avoir des gestionnaires différents en fonction du fil d'exécution qui reçoit le signal ? Le consensus actuel (ie. POSIX) est qu'il n'y a qu'une seule

série de gestionnaires d'interruption, mais que chaque fil d'exécution possède son propre masque de signaux. Si un signal peut être délivré à plusieurs fils d'exécution, il ne l'est qu'à un seul, au hasard. Un question annexe est ce que l'on peut faire dans un gestionnaire d'interruption. Peut-on réveiller un autre processus léger ? Peut-on commuter vers un autre processus léger ? Peut-on synchroniser le gestionnaire avec les processus légers ? Certains systèmes utilisent un processus léger pour réaliser la fonction du gestionnaire d'interruption ; d'autres l'implantent comme un véritable gestionnaire mais permettent sa transformation en un processus léger en cours de route [Stein & Shah 1992].

3.1.5.6 Gestion de la pile

La gestion de la pile peut être problématique. Les piles sont des zones de mémoire assez grandes. Leur taille exacte dépend du travail du fil d'exécution et peut difficilement être choisie par le support d'exécution. Soit l'utilisateur fournit une zone mémoire à utiliser comme pile, soit il indique la taille voulue et l'exécutif alloue la pile. Dans le second cas, il est possible de placer à la fin de la pile une zone rouge (*red zone*), indéfinie en mémoire virtuelle. Tout débordement de la pile sera alors piégé par le mécanisme de gestion de la mémoire virtuelle. Si les différentes piles sont organisées en mémoire virtuelle avec suffisamment d'espace libre entre, il est aussi possible de les étendre automatiquement. C'est ce qui est décrit pour les implantations sur Sun-LWP et Solaris [Sunsoft, Inc. 1992, Powell et al. 1991]. Cependant cela nécessite une participation du système et diminue la portabilité. D'autre part, puisque les piles sont assez grandes et donc que leur allocation est coûteuse, il est intéressant d'organiser un cache de piles, permettant une création plus rapide d'un nouveau fil d'exécution. Une autre solution, l'extension de pile, est proposée par le noyau Marcel (voir sa description en 5.2).

3.1.5.7 Gestion des contextes

Un dernier problème concerne la gestion du contexte d'un fil d'exécution. Au niveau matériel, il inclue les registres de l'éventuel coprocesseur mathématique ou autre. Ces registres doivent être sauvegardés lors de la commutation de contextes, ou tout au moins ceux qui sont en cours d'utilisation. L'utilisation de fonctions mathématiques peut lever un indicateur qui signalera la nécessité de sauvegarder les registres du coprocesseur. Au niveau logiciel, il est nécessaire de commuter les données contextuelles du processus léger, comme par exemple la valeur de la variable errno. Certains systèmes offrent une possibilité pour *définir* des données, au niveau du langage de programmation, dans une zone contextuelle en fonction du fil d'exécution [Powell et al. 1991] ; la plupart offrent simplement un mécanisme d'extension du contexte « système » du processus léger (par exemple, la primitive *pthread_keycreate* de POSIX).

3.1.6 Quelques noyaux de multiprogrammation légère

Nous présentons ici plus en détail quelques noyaux de multiprogrammation légère. Notre choix est loin d'être exhaustif ; nous présentons surtout les noyaux (généralement de niveau utilisateur) qui ont été utilisés dans l'implantation de notre exécutif. Nous procédons à une analyse comparative des différents noyaux en décrivant dans le même ordre chaque point qui nous a paru important. Un plus ample aperçu du domaine de la multiprogrammation légère est donné dans la section suivante.

3.1.6.1 Noyau « jouet » Briat (*Briat's Core*)

C'est un noyau implanté par J.Briat sur DG/UX, SunOS4.1 et Solaris 2.5. Il est utilisé pour l'enseignement. Il définit des fils d'exécution utilisateurs, avec un changement de contexte en assembleur. Les fils d'exécution sont préemptibles sur quantum de temps ; il n'y a pas de préemption sur

entrées-sorties ou de gestion des signaux. Les piles sont de taille fixe, la synchronisation se fait par sémaphores. Il n'y a pas de gestion de contextes. Les sources sont disponibles et c'est un squelette de multiprogrammation que les étudiants doivent enrichir (priorités, échéancier, entrées-sorties asynchrones).

3.1.6.2 REX

C'est un noyau implanté par Stephen Crane [Crane 1993] sur différentes architectures dont Linux, SunOS 4.1 (M68K et Sparc), Mips. Il définit des fils d'exécution utilisateurs, le changement de contexte est réalisé par setjmp() et longjmp() ; le noyau est donc « portable ». Il n'y a pas de préemption possible. Seuls les signaux SIGIO et SIGALARM sont utilisables. Ils sont normalement masqués, mais pris en compte lorsqu'aucun fil d'exécution ne tourne, ou bien tous les n changements de contextes, avec un certain surcoût sur l'opération de changement de contexte (jusqu'à 50% si n=1). Les gestionnaires d'interruptions sont commun à tous les fils d'exécution ; ils peuvent réveiller un fil d'exécution mais non commuter vers lui. Les fils d'exécution sont organisés selon des priorités (fixées lors de leur création). Les piles sont de tailles fixées à la création, la synchronisation se fait par sémaphores comptants. Une primitive permet d'associer un contexte à un fil d'exécution.

3.1.6.3 SunOS 4.1 LWP

Ce noyau [Sun Microsystems Inc. 1990, Sun Microsystems Inc. 1987*a*] est spécifique à SunOS 4.1 (M68K ou Sparc). Il définit des fils d'exécution utilisateurs, préemptibles. Les fils d'exécution sont gérés selon leur niveau de priorité, qui peut varier dynamiquement. Les gestionnaires d'interruptions sont partagés ; un signal est transformé en un message envoyé à un fil d'exécution particulier. Le fil d'exécution « simule » donc le gestionnaire d'interruption habituel. Une gestion synchrone des exceptions est aussi possible. Ce noyau définit un cache de piles (les piles allouées par lui sont réutilisées) ; de plus les piles sont extensibles et protégées par le mécanisme de la zone rouge. La synchronisation se fait avec des variables de condition et des moniteurs. Il est possible d'associer un contexte (pointeur) à un fil d'exécution. Les fils d'exécution peuvent converser par échange de message en plus de la mémoire partagée.

3.1.6.4 Solaris

Ce noyau [Eykholt et al. 1992, Sunsoft, Inc. 1992, Powell et al. 1991, Stein & Shah 1992] est intégré à Solaris. Il définit des fils d'exécution de niveaux noyau et utilisateur. La préemption est possible. Il y a un masque de signaux par fil d'exécution ; les signaux synchrones (exceptions) sont reçus par le fil d'exécution qui les génère, les signaux asynchrones sont reçus par n'importe lequel des fils d'exécution qui ne les masquent pas. Les gestionnaires d'interruption sont partagés. Les piles sont extensibles et protégées par une zone rouge. La synchronisation se fait par verrous, sémaphores comptants, variables de condition et protocole lecteurs / rédacteur. Une gestion évoluée des contextes est faite : le compilateur Sun comprends une directive spéciale (*unshared*) pour définir des variables contextuelles pour les fils d'exécution. Un espace leur est réservé avant le haut de pile. Un registre (implantation sur Sparc) est réservé par le compilateur pour pointer sur cette zone. Il est aussi possible d'associer plusieurs pointeurs de contexte à un fil d'exécution. L'utilisateur gère à la fois les fils d'exécution de niveau noyau et de niveau utilisateur. Il gère l'association des uns aux autres. Le noyau Solaris peut instancier de nouveaux fils d'exécution de niveau noyau, si nécessaire, pour éviter un interblocage. Il les réclame ensuite s'ils deviennent inutilisés. Les piles des fils d'exécution terminés sont libérées en tâche de fond quand un fil d'exécution noyau est libre. Plusieurs appels

systèmes ou fautes de pages sont possibles simultanément grâce aux fils d'exécution noyau. Notons qu'à l'intérieur du noyau Solaris, la plupart des interruptions sont manipulées comme des fils d'exécution noyau ; les interruptions peut être empilées... (la transformation en fil d'exécution n'intervient que si le gestionnaire d'interruption doit se bloquer). Un fil d'exécution noyau représentant des fils utilisateur a comme masque de signaux l'intersection des masques utilisateurs. Il peut donc recevoir tout signal et le délivrer à un fil utilisateur. De plus la manipulation du masque des signaux utilisateur se fait sans appel au noyau (si les fils utilisateurs se sont masqués entre temps, le fil noyau renvoi le signal à l'application, de sorte que d'autres fils noyau de l'application puissent le servir).

3.1.6.5 POSIX *Threads*

La norme POSIX [Institute of Electrical and Electronic Engineers, Inc. 1995] définit une interface de programmation pour les fils d'exécution. Plusieurs *drafts* ont été proposés successivement, le dernier (*draft 10 de 1003.4a*) devrait devenir la norme 1003.1c. Cette norme définit des fils d'exécution, implantés en général comme des fils de niveau utilisateur (la norme ne l'impose pas), préemptibles. Chaque fil d'exécution possède son propre masque de signaux. Les gestionnaires sont partagés, cependant. Les signaux sont délivrés de façon arbitraire aux fils d'exécution qui ne les masquent pas. Plusieurs politiques d'ordonnancement sont définies (FIFO, temps partagé, en tâche de fond...). Les fils d'exécutions sont gérés selon leurs priorités. La synchronisation se fait par variables de condition et verrous. Il est possible d'associer plusieurs contextes à un fil d'exécution (accessibles par une clé). Différentes implantations de cette norme sont disponibles, la plus connue étant celle de Mueller (FSU) [Mueller 1993, Mueller 1992].

3.1.6.6 DCE *Threads*

Cette implantation [Digital Equipment Corp. 1992] est conforme au *draft 4* de la norme POSIX. Elle définit des fils d'exécution de niveau utilisateur, préemptibles, avec des piles de tailles fixées. Elle est disponible sur Aix 3.2.5 et OSF/1 3.0.

3.1.6.7 MARCEL

Ce noyau , réalisé par l'équipe ESPACE pour l'exécutif PM^2, est disponible sur Sun OS 4.1 et Solaris (Sparc), OSF/1 3.0 (Alpha), Linux 1.3 (Intel) et maintenant Aix 3.2 (Power). Il est principalement conforme à POSIX. Il propose en plus un mécanisme d'extension de pile et de migration sur architectures homogènes (seule la pile du fil d'exécution est déplacée). Marcel est décrit plus en détail en 5.2.

3.1.7 Autres travaux sur la multiprogrammation légère

Les *C-threads* de MACH [Cooper & Draves 1988] sont probablement le premier noyau de fils d'exécution utilisateur « moderne ». Leur implantation était inspirée de MESA et de MODULA 2+. Ce sont des fils d'exécution de niveau utilisateur, implantés soit de façon coopérative (plusieurs C-threads pour un processus léger MACH), soit préemptive (un C- thread pour un processus léger MACH). La synchronisation se fait par verrous et variables de condition, il est possible d'effectuer des entrées-sorties asynchrones et il est possible d'associer un contexte à chaque fil d'exécution.

En réaction au défaut de cette implantation dans l'association C-thread / processus léger MACH, Levy [Anderson et al. 1992] propose la notion de *scheduler activations*, qui correspond approximativement à l 'association plusieurs fils d'exécution utilisateur / un fil d'exécution noyau de Solaris.

Il existe quelques différences cependant dans la façon dont l'association est gérée. La quasi- totalité des noyaux de système d'exploitation modernes (HP UX 9, IRIX 5, Aix 4, OS/2 Warp, Windows NT, Chorus [Armand et al. 1990], ...) offrent maintenant la notion de fils d'exécution noyau, en particulier pour soutenir les activités temps-réel et multimédia.

D'autres articles intéressants concernant la multiprogrammation légère sont indiqués ici :

- [Doeppner Jr. 1987, Anderson 1990, Goldstein et al. 1995, Keppel 1993*a*, Mukherjee et al. 1994, McJones & Swart 1987, Marsh et al. 1991] traitent de diverses questions d'implantation,
- [Draves et al. 1991, Ford & Lepreau 1993, Peacock et al. 1992, Tevanian et al. 1987, Dean 1993] s'intéressent à l'intégration de la mutilprogrammation légère dans divers systèmes d'exploitation,
- [Buhr & Stroobosscher 1990, Buhr et al. 1992, Sundaresan & Lee 1994, Giering & Baker 1992, Schmidtmann et al. 1993, Littman 1992, Shu n.d., Pinakis 1992] s'intéressent à l'utilisation des processus légers pour la réalisation de supports exécutifs variés,
- [Keppel 1993*b*] traite d'un noyau de très bas niveau permettant l'implantation efficace de diverses propositions de plus haut niveau,
- [Felton & McNamee 1992*b*, Felton & McNamee 1992*a*], [Rosing & Saltz 1993] et [Elmasri et al. 1994] traitent de noyaux de multiprogrammation légère pour l'amélioration des performances de communication,
- [Engler et al. 1993, Freeh et al. 1994*a*, Freeh et al. 1994*b*] traitent d'un noyau de multiprogrammation légère distribuée et
- [Wallach et al. 1995] utilise la multiprogrammation légère comme complément de la notion de message actif.
- [Kleiman & Eykholt 1995] montre l'approche retenue dans Solaris pour convertir une interruption en un processus léger.

Deux livres généraux sur la multiprogrammation légère sont particulièrement à recommander : [Kleiman et al. 1996] et [Birrell 1989]. On notera également l'article de synthèse [Demeure & Farhat 1994]. La thèse de J-N. Colin contient aussi un matériel intéressant [Colin 1995*b*]. Enfin, la bibliographie sur la multiprogrammation légère, disponible sur Internet à l'adresse *http ://liinwww.ira.uka.de/bibliography/os/threads.html*, permet d'approfondir ce tour d'horizon.

3.1.8 Conclusion

En conclusion, nous pouvons dire que les noyaux de multiprogrammation légère sont maintenant largement répandus et normalisés. Dans les faits cependant, chaque implantation est différente, en dépit de la normalisation de l'interface, à cause de la forte interaction de la multiprogrammation légère avec le reste du système d'exploitation d'une machine.

L'utilisation des noyaux de multiprogrammation, à l'époque du démarrage de cette thèse, était mature dans le domaine des machines SMP - des exécutifs à partage de mémoire - et des systèmes distribués - serveurs de fichier, etc... Cependant, dans le domaine du parallélisme sur machine à mémoire distribuée, seuls des fils d'exécution de niveau noyau étaient utilisés dans les systèmes d'exploitation parallèles. Les noyaux de fils d'exécution de niveau utilisateur n'ont quasiment pas été employés. L'un des intérêts de notre recherche est d'évaluer l'intégration d'un noyau de fils d'exécution de niveau utilisateur avec un noyau de communication par échange de messages. Citons trois propriétés importantes pour cette intégration :

Une opération de communication (par exemple la réception d'un message) sera dite **fil- sauve** (*thread-safe*) si elle peut être employée en concurrence par plusieurs fils d'exécution et donner un résultat correct pour tous, sans qu'il soit nécessaire d'effectuer une synchronisation. C'est le fait

d'être réentrant, mais vis à vis des fils d'exécution et non simplement vis à vis des interruptions (opération *signal-safe*). Deux niveaux peuvent être établis dans cette notion selon que des actions *indépendantes* ou *interdépendantes* sont correctement effectuées en concurrence. Par exemple, avec Solaris on peut faire plusieurs *read* sur des *sockets* différentes en concurrence ; par contre le résultat n'est pas garanti si l'on fait plusieurs *read* concurrents sur la *même socket*. Solaris est donc fil-sauf pour des actions indépendantes mais pas pour des actions interdépendantes. Nous prendrons comme définition commune de fil-sauf le premier niveau (vis à vis d'actions indépendantes, c'est la notion « *MT-safe* » de Solaris), et nous parlerons d'**opération fil-sauve sérialisée** si elle donne un résultat correct vis à vis d'actions interdépendantes.

Une opération sera dite **fil-synchrone** (*thread- synchronous*) si elle ne bloque que le fil qui l'exécute. En effet, une implantation fil-sauve courante d'une opération quelconque est basée sur l'exclusion globale : un seul fil a le droit d'exécuter l'opération à un moment donné. Le problème avec cette approche est la restriction de la concurrence qui en découle. A l'extrême, une opération peut bloquer non seulement le fil qui l'exécute, mais aussi toute l'application. C'est le cas par exemple d'une lecture bloquante du réseau si un seul processus est identifié par le noyau du système d'exploitation, car le blocage se produit à l'intérieur de celui-ci.

Enfin une bibliothèque de communication sera qualifiée du terme « **système reconnaissant la multiprogrammation légère** » (*thread-aware system*) si elle peut gérer explicitement des informations ou a un comportement dépendant de la multiprogrammation légère. C'est le cas par exemple si la bibliothèque en question peut réveiller un fil d'exécution particulier sur un événement donné, comme l'arrivée d'un message.

3.2 Un noyau de communication : Parallel Virtual Machine

Le prototype d'Athapascan-0 ayant été réalisé au dessus de PVM, nous avons choisi de décrire en détail cet outil comme exemple d'un noyau de communication sur machine parallèle à mémoire distribuée. Nous présentons les deux implantations de PVM que nous avons utilisé : la version domaine public et la version propriétaire d'IBM, PVMe.

3.2.1 Présentation de PVM

Le but de PVM (*Parallel Virtual Machine*) [Sunderam 1990, Dongarra et al. 1993, Geist et al. 1993, Geist et al. 1994, Manchek 1994, Grant et al. 1992] est d'utiliser un ensemble de machines interconnectées comme une unique machine parallèle virtuelle. L'intérêt réside dans le fait que la machine résultante est extensible (par adjonction de machines physiques), modifiable (par remplacement standard de composants périmés), stable (car elle s'appuie sur une technologie de masse qui est au point), qui s'utilise dans un environnement habituel (celui d'un laboratoire, Unix) et qui peut comporter des modules spécialisés (stations graphiques, processeurs vectoriels, ...). De plus PVM permet d'utiliser un réseau existant, ou bien de relier plusieurs machines parallèles entre elles, de façon à produire une machine virtuelle de puissance raisonnable pour un coût relativement faible. En contrepartie, il faut s'abstraire du problème d'hétérogénéité, à la fois au niveau du format des données qui peut différer d'un processeur à l'autre, mais aussi en ce qui concerne la différence de vitesse ou de charge des machines, qui si elle n'est pas prise en compte peut grever l'efficacité globale du système en sous-utilisant les machines les plus rapides. PVM propose un modèle de programmation par échange de messages entre tâches, assez souple puisque n'importe quelle tâche peut interagir avec n'importe quelle autre, et même en créer de nouvelle, et offre principalement une gestion des tâches,

une gestion des messages et une conversion automatique des formats de données hétérogènes. Par contre, l'utilisateur doit gérer lui-même le partitionnement et l'ordonnancement de son programme.

La version 1 de PVM a vu le jour en 1989 à l 'Oak Ridge National Laboratory. Cette version, de recherche, n'a pas été distribuée. La version 2 fut distribuée en 1991 par l'Université du Tennessee. La version 3, disponible depuis 1993, manifeste un changement d'interface, offrant entre autre la possibilité nouvelle d'utiliser plusieurs tampons de communication simultanément. Cette version, disponible au début de ce travail, fut principalement utilisée. Le tableau 3.1 montre l 'ensemble des machines sur lesquelles PVM 3 a été porté. On remarquera un certain nombre de machines parallèles à mémoire distribuée ou partagée, en plus des stations de travail habituelles. PVM demande de simple possibilités de multiprogrammation lourde et de mise en réseau et a donc aussi pu être implanté sur des systèmes assez éloignés d'Unix comme OS/2 [2] ou les moutures de MS- Windows [3] . D'autre part, les vendeurs de machines parallèles ont souvent développé leur propre « clone » de PVM, efficacement implanté sur le noyau de communication natif (propriétaire). Outre IBM avec PVMe, nous pouvons citer Cray pour le T3D, et DEC pour le DEC2100. Enfin, PVM peut se définir comme une sur- couche au dessus de MPI, facilitant la gestion de la machine virtuelle (qui n'est pas actuellement présente dans MPI).

Alliant FX/8	Dec Station 3100, 5100	Intel Paragon	Silicon Graphics IRIS
BBN Butterfly TC2000	Encore 88000	Kendall Square KSR1	Sparc Multiprocessor
C-90, Cray YMP, T3D	HP9000/300	Maspar	Stardent Titan
Convex C-series	HP9000/PA-RISC	Mips 4680	Sun 3
Cray S-MP	IBM 370	NeXT	Sun 4, SparcStation
Cray-2	IBM RS6000	PC ix86	Thinking Machine CM2
Data General Aviion	IBM RT	Sequent Balance	Thinking Machine CM5
Dec Alpha	Intel iPSC/2	Sequent Symmetry	
Dec MicroVAX	Intel iPSC/860	SGI Multiprocessor	

TAB. 3.1 - *Les portages de PVM.*

Le tableau indique les différentes architectures supportées.

3.2.2 Description de PVM

PVM définit des tâches (en général des processus Unix) qui vont communiquer par l'intermédiaire d'une bibliothèque de fonctions offrant une certaine *API, Application Programming Interface*, et de démons servant principalement de routeurs dans la communication. Une tâche spéciale, la console, permet à l'utilisateur d'interagir avec la machine virtuelle (l'étendre, l'arrêter, démarrer une application, tester son état,. . .).

Les fonctions de l'API peuvent se classer comme suit :

- Configuration dynamique de la machine virtuelle : ajout et retrait de nœuds physiques
- Contrôle de processus : s'enrôler dans ou bien quitter la machine virtuelle, démarrer une tâche ou la tuer
- Information : identification de la tâche parente, information sur la configuration de la machine et identification des tâches actives
- Signalisation : transmission de signaux Unix, notification d'événements concernant la machine

2. ftp ://ftp- os2.cdrom.com/pub/os2/2_x/program/pvm3os2b.zip
3. http ://netlib2.cs.utk.edu/pvm3/pvm_win32.zip et http ://netlib2.cs.utk.edu/pvm3/pvm_win32.tar.gz

- Echange de messages : gestion de tampons de communication, emballage et déballage de données typées, émission et réception de messages étiquetés
- Gestion de groupes dynamiques : nommés (gérés par un gestionnaire spécifique), construits sur les fonctionnalités propres de PVM. Les groupes sont ouverts et dynamiques, souplesse d'utilisation qui se paye en terme d'efficacité et de précision sémantique. Les fonctions sont l'enrôlement dans, la désertion d'un groupe, l'information sur les composants d'un groupe, l'identification d'un composant au sein d'un groupe, la synchronisation (*barrier*), la diffusion (*broadcast*), la réduction (par une opération définie par l'utilisateur ou prédéfinie)

Le modèle de messages mis en œuvre est une modèle semi-synchrone (on n'attend pas que le correspondant ait posté l'opération correspondante), bloquant (en revanche on attend que le tampon soit à nouveau disponible). L'ordre des messages entre deux processeurs est préservé (ordre *FIFO*) par défaut. Cependant, il est possible de recevoir un message hors séquence en spécifiant une valeur d'étiquette particulière. Tous les messages ne présentant pas la bonne étiquette sont tamponnés sur le site destination, en vue de leur réception ultérieure. Il est aussi possible de filtrer les messages par leur site d'origine. Il n'y a pas de contrôle de flux (les messages sont tamponnés à la réception, jusqu'à explosion de la mémoire virtuelle). Enfin la taille maximale d'un message est variable, moins de la moitié de la taille de la mémoire virtuelle disponible.

Les données sont emballées dans les messages en spécifiant une liste de blocs typés, éventuellement parsemés, discontigüs. Elles sont déballées de la même façon.

Les primitives d'émission et de réception sont :

- l'émission point à point bloquante,
- l'émission multipoints bloquante,
- la réception bloquante ou non-bloquante, avec temps limite éventuel, ou bien seulement test de possibilité de réception sans blocage.

Il y a aussi des primitives pour emballer-et-envoyer (et déballer-et-recevoir) principalement utiles pour les multiprocesseurs, car elles correspondent plus directement aux primitives de communication natives disponibles, et évitent le surcoût d'un emballage séparé. Ces primitives ont été ajoutées dans les dernières versions de PVM.

Les taches sont identifiées à l'aide d'un identificateur sur 32 bits, contenant le numéro du nœud dans la machine virtuelle (jusqu'à 4096 processeurs physiques) et le numéro de la tâche sur le nœud (jusqu'à 256000 tâches par nœud). Dans le cadre d'un multiprocesseur, ce dernier nombre peut être raffiné en un numéro de processeur et un numéro de tâche sur le processeur. L'identification d'une tâche ne nécessite donc pas d'opération globale, ni à sa création (puisque c'est le processeur physique local qui attribue un numéro de tâche), ni à son utilisation (puisque l'adresse du processeur physique où se trouve la tâche est contenue dans son identification).

PVM offre des fonctionnalités importantes :

- tolérance aux pannes limitées : en résolvant les erreurs dues au réseau ou aux nœuds et en en informant l'application si nécessaire. Cependant l'application doit prendre en compte elle même les pannes importantes,
- extensibilité : en utilisant une gestion décentralisée et localisée,
- hétérogénéité de formats de données : par l'intermédiaire de XDR [Sun Microsystems Inc. 1987*b*] et de messages typés,
- portabilité : en utilisant les fonctionnalités standard d'Unix (multiprogrammation lourde, mise en réseau). En particulier, les fonctions de multiprogrammation légère et d'entrées / sorties asynchrones ne sont pas employées car elles n'ont pas été jugées suffisamment unifiées et répandues.

Dans la suite, nous allons détailler plus précisément le fonctionnement des démons et surtout la gestion des messages.

3.2.3 Détail du fonctionnement de la version Domaine Public

3.2.3.1 Les démons

Il y a un démon par nœud. Il sert de routeur et de contrôleur :

- il offre un point de contact bien défini sur la machine
- il prend en chargé l'authentification
- il contrôle les processus (tâches) sur le nœud
- il détecte les pannes en testant les correspondants de temps en temps
- il reste présent si l'application plante, facilitant de ce fait le déverminage

Le premier démon (appelé *master*) est le seul qui puisse ajouter / enlever des nœuds à la machine virtuelle. Il gère en effet une table de nœuds, qu'il partage avec les autres démons (simplement en signalant les retraits de nœuds, mais en utilisant un protocole à trois phases pour les ajouts de nœuds, de façon à ce que tous gardent une vision cohérente de la machine virtuelle). Ce démon maître est une faiblesse dans la tolérance aux pannes ; en effet, bien que tout démon esclave puisse mourir inopinément, le démon maître doit rester vivant et accessible des autres, puisqu'il gère la table des nœuds.

Chaque démon gère une table des tâches locales. Une tâche locale peut s'enrôler dans la machine virtuelle, auquel cas elle contacte le démon local suivant les informations indiquées dans un fichier spécial, ou bien peut être démarrée par le démon local si elle est « *spawnée* » depuis une application. Dans ce dernier cas, elle va se connecter (se reconnecter) au démon qui l'a créée. Dans l'intervalle, les messages à destination de la tâche sont tamponnés dans le démon.

Le démon effectue aussi une authentification des tâches qui se connectent, de façon à vérifier que l'utilisateur n'est pas trompé par un intrus. Symétriquement, les tâches font aussi cette identification, pour vérifier la légitimité du démon.

Un protocole particulier (*startup protocol*) est mis en œuvre lors de la création d'un démon esclave. En effet, c'est un processus assez long. C'est pourquoi un « *shadow demon* » est utilisé pour mener à bien cette opération, laissant le démon maître libre de brasser les communications en cours. Le *shadow demon* utilise en général une commande *rsh* pour démarrer le démon esclave, puisque c'est un dispositif hautement portable.

Bien que des mécanismes par défaut soient définis, il est possible de les redéfinir à travers une interface simple, en ce qui concerne :

- l'ajout et le retrait d'une machine,
- la sélection d'une machine pour y faire tourner une tâche, l'information sur la configuration des tâches, la signalisation,
- le démarrage d'une tâche.

Ceci se fait en définissant trois tâches particulières, le *hoster*, le *resource manager* et le *tasker*. Les démons court-circuitent alors le mécanisme par défaut et communiquent avec ces tâches pour effectuer les fonctions demandées. Ceci permet une meilleure adaptabilité à un environnement particulier, et aussi permet d'effectuer une régulation de charge complexe lors du démarrage des tâches.

3.2.3.2 La gestion des messages

Les tâches ne communiquent pas directement entre elles, en général. Elles passent par le démon local, qui à son tour va contacter le démon de la tâche distante, qui lui-même contactera la tâche destinatrice. En effet, une tâche devant effectuer un calcul et ne disposant pas d'entrées-sorties asynchrones, ne peut pas utiliser de façon efficace un protocole non fiable comme UDP [Postel 1981*b*]. Utiliser un protocole spécialisé comme VMTP [Cheriton 1988] aurait requis la modification des noyaux Unix et aurait donc grevé la portabilité. Un protocole fiable, TCP, est donc employé, mais la limitation courante du nombre de descripteurs de fichiers ouverts fait que cette méthode n'est pas directement extensible, puisqu'elle nécessite un descripteur de fichier par lien TCP et donc par processeur virtuel distant. Les démons, s'intercalant entre les tâches, peuvent utiliser un protocole non fiable entre eux, puisque leur tâche principale est de communiquer. Le schéma suivant résume la situation. Les démons servent donc principalement de routeurs pour les messages.

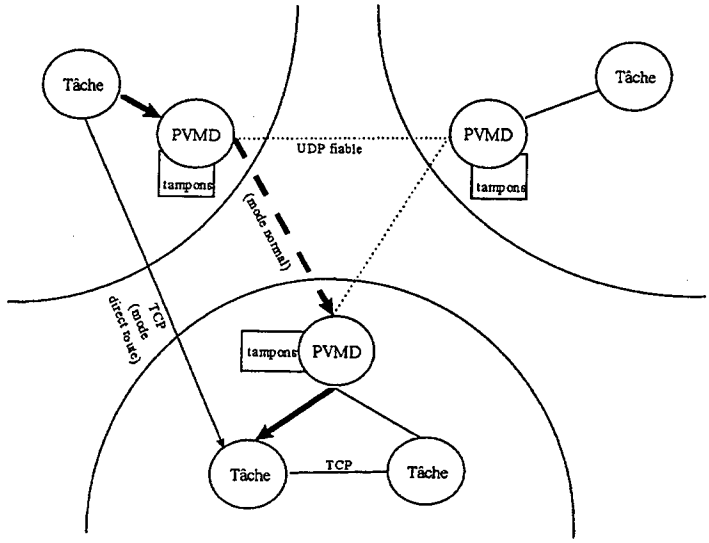

FIG. 3.1 - *Circulation des paquets dans PVM.*

Le mode normal de circulation passe sur le même site par un lien TCP et à distance par les démons PVMD. Le mode direct-route emprunte des liens TCP même à distance, mais restreint le nombre de tâches.

Si la limitation du nombre de descripteurs de fichiers n'est pas un problème, une tâche peut se connecter directement à une autre tâche grâce à une liaison TCP (c'est le mode « *direct route* »). Le fait d'éviter une communication à travers les démons double approximativement le débit. Ce mécanisme est utilisable à tout moment et permet de relier certaines tâches entre elles sélectivement. Si les descripteurs de fichiers sont épuisés, le chemin à travers les démons est obligatoire (et utilisé de façon transparente).

Les liens tâche-démon sont en général effectués sur des *sockets Unix-domain* et non pas *inet*. Cela accélère la communication quand c'est possible. Pour gérer l'hétérogénéité de format de données, les messages sont typés et encodés / décodés. Les encodages possibles sont :

- *raw* : les formats de données sont homogènes (pas d'encodage)
- *default* : on applique XDR
- *foo* : un format XDR simplifié pour la communication vers / depuis les démons
- *alien* : la donnée ne peut être décodée, seulement retransmise

- *inplace* : lors de l'emballage, les données sont laissées en place, seul le pointeur sur la donnée est mémorisé (cela évite un copie). La donnée sera décodée par *raw* ou *default* (XDR). Ce mode sera appelé **transfert en place** dans la suite.

Comme le protocole UDP impose une taille maximum pour les paquets qu'il transmet, les messages sont fragmentés, les paquets routés à travers les démons, puis rassemblés. Un démon boucle donc sur une primitive Unix *select* de tous les liens qu'il possède (TCP vers les tâches locales, UDP vers les autres démons) et choisit un lien prêt à communiquer : envoyer ou recevoir. Selon la destination finale d'un paquet reçu, soit ce paquet est dirigé vers les files d'émission, soit il est assemblé en un message, puis traité par l'un des trois points d'entrée du démon :

- *locentry()* pour gérer les messages en provenance des tâches locales,
- *netentry()* pour gérer les messages en provenance des autres démons,
- *schentry()* pour gérer les tâches spéciales (*hoster, tasker, ...*).

3.2.3.3 Protocole pour UDP : démon à démon

Les paquets à émettre sont gérés dans une file par nœud destinataire. Le protocole UDP n'étant pas fiable, les paquets émis sont mis en file (par nœud destinataire) tant qu'il n'ont pas été acquittés. Les paquets reçus hors séquence sont maintenus dans une liste. Une file triée par date de réémission, globale pour tous les nœuds destinataires, est aussi maintenue. Le délai de d'aller-retour (*roundtrip delay*) entre émission et acquittement est mesuré dynamiquement (calculé avec un fondu des précédentes valeurs) et les paquets non acquittés sont réémis selon un temps progressif double à chaque réémission, basé sur le délai d'aller-retour. Les paquets peuvent être réémis jusqu'à dix fois, ou pendant trois minutes après la première émission ; au delà, le nœud destinataire est considéré comme mort.

Dans l'autre sens, les paquets destinés au démon sont rassemblés (par nœud d'origine).

Le démon doit pouvoir gérer plusieurs communications en parallèle. Il utilise donc un « *wait context* » (point de reprise) qui contient le contexte en cours de la communication. Ce *wc* est identifié par un numéro (en séquence) et mis en file triée (par numéro). Ce *wc* est transmis lors d'une requête et transmis en retour avec la réponse à la requête. En conséquence, l'opération temporairement mise en attente peut être continuée. Si plusieurs opérations doivent être faites en parallèle, plusieurs *wc* sont établis, et liés entre eux. Quand le dernier *wc* de la liste est traité, l'opération est terminée.

Chaque message commence par un entête de 16 octets, contenant l'étiquette, le type d'encodage, un identificateur de *wc* et un checksum (sur le premier paquet). Chaque paquet d'un message contient aussi un entête de 16 octets : les identificateurs de tâches source et destination, le numéro dans la séquence, le numéro d'acquittement et des indicateurs. Les messages sont envoyés dans l'ordre, paquet après paquet (par nœud). Un message devant être *multicasté* n'est présent physiquement qu'une fois en mémoire. Il est référencé dans plusieurs queues (une fois par nœud destinataire). Un compte de référence est maintenu. Quand le compte descend à zéro, le paquet est effacé.

3.2.3.4 Protocole pour TCP : entre démon et tâche

Au dessus de TCP est reconstruit un service de datagrammes. Les limites des messages sont distinguées par des indicateurs dans l'en-tête du paquet. La longueur de chaque paquet est mise dans l'en-tête, pour distinguer les paquets. TCP étant fiable, il n'y a pas de liste de paquets à réémettre.

Donc les tâches découpent les messages en paquets à l'intérieur de la bibliothèque PVM. Inversement, elles les rassemblent en messages, avec toujours un seul message en cours de construction.

Les démons routent en général les paquets tels quels, mais peuvent les assembler s'ils sont destinés à eux même. Il est aussi possible qu'un paquet reçu soit trop grand pour être réémis tel quel. Il est alors fragmenté à la bonne taille, en générant de nouveaux en-têtes qui pointent sur le paquet initial.

Dans un tâche, l'évolution des communication se fait lors de *mxfer()*, primitive qui est utilisée lors des opérations *pvm_send, pvm_recv,* ou *pvm_nrecv...*

3.2.3.5 Portabilité - Adéquation aux multiprocesseurs

PVM utilisant des fonctionnalités standard d'Unix, la portabilité vers toute machine monoprocesseur pourvue d'un système d'exploitation proche d'un Unix complet standard n'est pas trop problématique. Par contre, la portabilité vers des *clusters* de stations pourvues d'un réseau rapide, ou vers un multiprocesseur à mémoire partagée, est plus compliquée. Les portages vers les *clusters* de stations posent deux problèmes principalement : la gestion des tâches doit être revue puisque bien souvent s'intercale une notion de partitionnement de la machine *cluster* (souvent « statique »), et que le chargement d'un binaire peut être différent des mécanismes Unix traditionnels. D'autre part, l'efficacité du réseau de communication fait que, même si les primitives de PVM se carrossent relativement facilement sur les primitives natives, le mécanisme d'emballage des données n'est pas performant. De nouvelles opérations *pvm_psend* et *pvm_precv*, pouvant être implantées plus facilement et efficacement sur un *cluster*, ont donc été rajoutées dans les dernières versions. C'est là une limite de la portabilité du modèle PVM initial. En ce qui concerne le multiprocesseurs à mémoire partagée, il est nécessaire de transcrire les primitives de passage de messages en primitives agissant sur la mémoire partagée. En général, un segment de mémoire partagée est alloué, et chaque fragment de message est copié dans une partie de ce segment, puis la tâche destinatrice est informée de l'adresse qu'elle doit aller lire, grâce à une boîte aux lettres dans son propre segment. La tâche destinatrice indique qu'elle a lu le fragment en modifiant celui-ci. La synchronisation se fait avec un verrou établi en mémoire partagée, principalement lors de l'écriture dans la boîte aux lettres destinatrice.

3.2.3.6 Divers

PVM offre en plus des facilités agréables pour le programmeur par exemple la possibilité de transmettre un morceau d'environnement à la tâche créée, la possibilité de rediriger les *stdout / stderr* des tâches filles, celle de lancer les tâches filles sous le contrôle d'un dévermineur et de tracer l'exécution de PVM (encore qu'avec une forte perturbation sur le comportement de l'application, puisque les traces circulent sur le médium de communication en temps réel).

3.2.3.7 Conclusion

Nous avons présenté ici rapidement PVM version domaine public. Le lecteur trouvera de plus amples informations dans le *master thesis* de B.Mancheck [Manchek 1994], ainsi que dans le *PVM Book* [Geist et al. 1994]. En guise de conclusion, nous pouvons dire que PVM est un système souple ; porté sur un grand nombre de machines, qui est relativement extensible et qui accepte des configurations hétérogènes. Cependant un certains nombre de défauts graves peuvent être cités :

- Le modèle de PVM est un modèle dynamique mais lourd : la création d'une tâche correspond à la création d'un processus Unix. D'autre part, le modèle d'interaction (échange de messages entre processus préexistants) implique la mise en œuvre d'un automate de communication, séquentiel. Le modèle de PVM n'est donc pas très adapté aux applications parallèles irrégulières.

- PVM ne fournit pas d'interruption sur réception d'un message. En conséquence, il est nécessaire de scruter le réseau pour savoir si un message est arrivé. L'intervalle entre chaque scrutation définit le temps de réponse de l'application. Il est difficile de garantir ce temps de réponse.
- Le modèle de PVM est un modèle à tampon infinis : les messages sont tamponnés de façon non synchrone. La mémoire virtuelle est utilisée pour stocker les messages en attente. En conséquence, l'application peut épuiser la mémoire virtuelle à tout moment, de façon difficilement contrôlable. Il en résulte un risque d'erreur indéterministe. En pratique, beaucoup d'applications présentent suffisamment de synchronisme pour que ce cas de figure se produise rarement. En particulier les applications régulières ne posent guère de problèmes.

Des défauts moins importants sont aussi présents :

- PVM est réputé *thread-unsafe*. En effet, PVM gère de façon globale les tampons de communication. La préemption d'un fil pollue l'exécution si l'on ne prend pas garde à sauvegarder les identités des tampons utilisés pour les restaurer plus tard. Cependant, ce mécanisme de sauvegarde / restauration est facile à mettre en œuvre, particulièrement lorsque les fils d'exécution sont coopératifs et que l'on garde le contrôle des commutations, puisque l'API de PVM version 3 possède une propriété très intéressante pour la multiprogrammation : il est possible d'emballer ou de déballer plusieurs messages en parallèle, en commutant entre plusieurs tampons de communication.
- L'implantation domaine public n'est pas très efficace : nous avons vu qu'elle présente un nombre important de copies (dans la tâche, des données réelles vers les tampons, des tampons vers la couche pilote réseau, deux fois du pilote réseau vers les tampons vers le pilote réseau dans les démons, et encore du pilote réseau vers les tampons vers les données réelles dans la tâche destinatrice, soit huit copies en tout), l'émission de nombreux paquets d'acquittement (un pour un paquet envoyé) et la fragmentation des données sur TCP, ce qui n'est peut être pas obligatoirement nécessaire.
- La réalisation *cluster* a nécessité de casser le modèle initial (emballage / déballage) pour obtenir une réalisation efficace, et la réalisation sur mémoire partagée implique aussi un certain nombre de copies...
- Il est impossible de modifier des données déjà emballées dans un tampon de communication. En conséquence, l'entête des messages doit être formé avant les données. Cet entête ne peut pas contenir par exemple, la taille des données, si celle-ci doit être calculée au fur et à mesure de l'emballage. Ce comportement complique un peu la réalisation d'extensions, aussi bien au dessus de PVM, qu'au dessus d'un outil qui utilise PVM, comme Athapascan.
- Nous avons vu que PVM effectue automatiquement les reprises d'erreur. En particulier, si une tâche ne peut être contactée, la réémission des messages est automatique. Cependant, au bout d'un certain nombre d'essais la tâche est déclarée inaccessible ; elle est bien retirée de la liste des tâches, mais aucun retour d'erreur n'est effectué pour les messages déjà envoyés. La tolérance aux pannes est donc très limitée.

Les avantages de PVM sont que le modèle est souple d'utilisation et offre une gestion de la machine virtuelle. On peut donc dire que la souplesse d'utilisation et la disponibilité ont une grande plus importance dans PVM et que l'efficacité brute ou la rigueur sémantique.

3.2.4 La version spécifique d'IBM, PVMe

PVMe [IBM 1994, Bernaschi & Richelli 1995] est une implantation spécifique de l'API de PVM[4] pour les machines IBM SPx. Elle tire parti du réseau rapide de l'IBM, « l'*HighPerformanceSwitch* », en utilisant directement la bibliothèque *CSS-CI* d'interface avec l'adaptateur réseau, et non *IP*, comme le fait PVM domaine public. Cette bibliothèque place l'adaptateur réseau dans l'espace mémoire du processus ; en conséquence les données peuvent directement transiter sur le réseau sans copie intermédiaire.

Dans PVMe, il n'y a qu'un démon qui gère la base de données des tâches et qui est responsable de créer, terminer et synchroniser les tâches. Les tâches envoient directement les paquets des messages. Une entité particulière, le *reader*, reçoit les messages pour le compte d'une tâche. Ce reader gère le contrôle de flux. Il peut être implanté comme un processus distinct (communicant par mémoire partagée) ou comme un gestionnaire d'interruption à l'intérieur de la tâche utilisateur. Le nombre de copie des messages est réduit à deux sur le récepteur, plus une sur l'émetteur si l'option *DataInPlace* n'est pas employée. Le but du *reader* est principalement de recevoir les messages et de les stocker quelque part, en attendant que le programme utilisateur veuille bien les lire. La technique d'allocation d'espace pour les messages en attente est optimisée par une allocation directe en mémoire virtuelle, sous le contrôle de PVMe.

En conclusion PVMe est une implantation plus efficace du modèle de programmation de PVM, pour l'IBM SPx. Une comparaison des performances entre PVM domaine public et PVMe est donnée 4.7.4. L'intérêt d'utiliser PVMe sur le SP1 est flagrant.

3.3 Conclusion

Dans le précédent chapitre, nous avons vu qu'il faut un exécutif particulier pour supporter les applications parallèles irrégulières. Un tel exécutif portable n'était pas répandu à l'époque du début de cette thèse et est absolument nécessaire à tout projet visant le développement d'un environnement de programmation pour de telles applications. Actuellement de nombreux projets de recherche s'intéressent à ce problème (voir chap. 5). Nous avons examiné en détail les mécanismes possibles pour un support exécutif de ce type. Il s'agit de la multiprogrammation légère et de la communication par échange de messages sur machines à mémoire distribuée.

Nous avons vu divers degrés de multiprogrammation légère : de niveau utilisateur, de niveau noyau ou mixte ; coopérative ou préemptive sur temps partagé ou encore sur entrées-sorties.

De même, nous avons vu que PVM est un outil intéressant mais pas complètement adéquat ; en particulier sa dynamique lourde et le fait que l'on doive construire un automate ne favorisent pas son utilisation pour les applications irrégulières.

Dans la suite nous allons montrer comment l'intégration de la multiprogrammation légère dans PVM peut conduire à un support exécutif adéquat pour les applications irrégulières. En fait, la base est déjà présente dans la structure des démons de PVM domaine public. Cette structure est, grosso modo, une boucle infinie sur une primitive *select* qui fait progresser les communications (envoi ou réception de fragments). Lorsque suffisamment de fragments sont reçus pour constituer un message complet, l'un des trois points d'entrée du démon est invoqué. Il est nécessaire de multiplexer le fonctionnement des points d'entrées avec la boucle principale de progression des communications, en

4. La version 1.2 de PVMe est compatible avec l'interface 3.2 de PVM domaine public ; pour Athapascan-0a nous avons utilisé les versions 3.3.x de PVM domaine public, mais les différences dans l'API - hormis ce qui concerne les groupes, qu'Athapascan n'utilise pas - sont suffisamment mineures pour ne pas poser de problème.

particulier lorsque l'exécution d'un point d'entrée aboutit à des communications avec les autres démons ou tâches. Le mécanismes des points de reprise (*wait contexts*) que nous avons décrit permet de réaliser un coroutinage entre les points d'entrée et la boucle principale de l'automate. C'est en quelque sorte un noyau de multiprogrammation légère simplifié (les contextes commutés sont totalement identifiés dans les points de reprise). Programmer à ce niveau de détail système n'est pas approprié pour le spécialiste du calcul numérique. Tout l'intérêt d'un système de programmation simple et efficace, qui intègre multiprogrammation légère et communications en vue de la réalisation d'applications irrégulières est de réaliser cette tâche automatiquement.

Chapitre 4

Réalisation et évaluation d'un support d'exécution pour applications parallèles irrégulières

Dans la suite nous allons présenter l'implantation du prototype d'Athapascan-0 sur PVM, implantation nommée Athapascan-0a. Nous abordons les objectifs de cette implantation, les choix de conception que nous avons pris, l'organisation retenue, le modèle et l'interface de programmation qu'elle supporte. Nous mentionnons les fonctionnalités requises pour les noyaux de multiprogrammation et le choix du noyau de communication qui nous a conduit à PVM, avant de décrire l'implantation réalisée. Puis nous analysons les performances de cette implantation en comparaison à celles des communications natives, dans le but de mettre en évidence le surcoût introduit. Nous terminons par une conclusion sur Athapascan-0a qui fixe les limites de ce prototype.

4.1 Les objectifs d'Athapascan-0a

Nous résumons dans un premier temps les objectifs généraux d'un exécutif parallèle puis nous indiquons les objectifs particuliers de ce travail qui nous ont conduits à Athapascan-0a.

Les objectifs d'un support exécutif parallèle peuvent être classés selon deux grandes catégories : l'expressivité et l'efficacité. Nous détaillons différents aspects de ces deux catégories. Un exécutif parallèle doit être :

(Expressivité)

- *Expressif* - L'exécutif parallèle doit être à même de supporter toute application parallèle, avec la restriction du modèle conceptuel dans lequel il a été conçu. Il doit donc offrir suffisamment de constructeurs pour exprimer les cas d'école d'une application dans ce modèle. Cependant ces constructeurs ne doivent pas forcément être complexes ; ils peuvent rester élémentaires même si c'est au dépend de la facilité d'utilisation de l'exécutif. En effet, l'exécutif peut et doit normalement être la cible d'une couche de plus haut niveau, compilateur ou bibliothèque, qui sera à même d'offrir la facilité d'utilisation au programmeur. L'expressivité - la complétude - des constructeurs de l'exécutif garantit la possibilité de définir une couche d'interface avec l'utilisateur qui sera elle-même complète.
- *Versatile* - L'exécutif est la couche de base souple, portable, efficace. Cependant ce n'est pas la couche finale de l'environnement de programmation parallèle. Il est donc nécessaire de pouvoir facilement construire au-dessus de cette couche de base, par exemple une couche de régulation de charge, un compilateur, une bibliothèque de fonctions...

- *Structurant* - L'exécutif doit proposer des opérateurs favorisant la construction structurée et modulaire d'une application. Il doit être à même de permettre la réutilisation de composants d'applications, en vue de réduire les coûts de développement. Cette modularité dépend bien sûr du compilateur ou de la bibliothèque définis au-dessus de la couche exécutive, mais l'exécutif en lui-même peut grever la modularité souhaitée.
- *Simple et défini* - Une sémantique précise et simple permet de développer, de maintenir et d'utiliser plus facilement un composant logiciel. De même, l'éradication des bogues à tous niveaux en est favorisée.

(Efficacité)

- *Portable* - Economiquement parlant, une application parallèle se doit d'être portable sur une grande variété de machines parallèles. L'exécutif qui supporte l'application doit donc être portable ; de plus il ne doit pas simplement être facilement portable, il doit aussi conserver son efficacité et l'efficacité des applications parallèles développées sur lui. Nous avons vu que l'exécutif à lui seul ne peut pas garantir l'efficacité de l'application sur une machine donnée ; par contre il peut renseigner l'application sur les indicateurs d'efficacité d'exécution.
- *Efficace* - Bien que l'efficacité brute d'un exécutif soit souvent mise en exergue dans les publications, nous pensons qu'économiquement, ce n'est pas la priorité. Les machines parallèles se succèdent suffisamment vite pour que l'efficacité voulue soit atteinte assez rapidement. En revanche, l'expressivité et la portabilité sont deux objectifs plus importants puisqu'ils garantissent la pérennité d'un investissement. Cependant, l'efficacité obtenue doit rester proche de celle du système natif. Pour les langages séquentiels, un facteur 4 entre l'efficacité du code compilé et celle du code assembleur est souvent considéré comme la limite acceptable.
- *Extensible* - La faculté de s'exécuter sur des machines fortement parallèles sans faire apparaître de grands surcoûts de gestion est l'un des aspects de la portabilité efficace.
- *Observable* - L'observabilité que donne l'exécutif de son fonctionnement et de celui de l'application est un point important dans l'apprentissage que peut faire l'utilisateur sur ceux-ci. Une meilleure observabilité aboutit à de meilleurs programmes. Dans la même catégorie nous rangerons un certain déterminisme de l'exécutif.
- *Commandable* - Grâce aux renseignements fournis par l'exécutif sur son fonctionnement, l'application peut être à même de décider d'un changement dans son mode d'exécution. La commandabilité de l'exécutif permet à l'application de rétroagir sur son fonctionnement, de façon à s'adapter au mieux à l'environnement.
- *Tolérant aux pannes* - Comme dans un système distribué, la tolérance aux pannes est importante. L'exécutif doit être à même, soit de corriger silencieusement les pannes mineures, soit de rapporter les pannes majeures. Dans tous les cas son fonctionnement doit être le plus assuré possible, même en environnement perturbé. Toutefois l'application peut décharger l'exécutif d'une partie de la tolérance aux pannes de l'application elle-même. La tolérance aux pannes se fait généralement au détriment de l'efficacité globale.

Dans le cadre d'APACHE et de cette thèse, l'objectif principal était de *réaliser et d'évaluer un prototype d'un support d'exécution*, dénommé Athapascan-0a, pour juger de la validité de l'approche et permettre aux autres travaux (régulation de charge, réexécution déterministe, prise de traces, visualisation..) de démarrer avec un support cible existant.

Les objectifs secondaires étaient :

- de tester différentes approches pour un tel support d'exécution,
- d'assurer une efficacité raisonnable de façon à pouvoir afficher des résultats effectifs,
- d'assurer une certaine expression de la programmation parallèle en fonction d'un modèle de programmation décidé au préalable,

- d'assurer la portabilité en prévision de l'achat d'une machine parallèle non encore choisie,
- et de réserver l'extensibilité, l'observabilité et la maniabilité du système pour des expérimentations futures.

4.2 Choix de conception pour Athapascan-0

Nous avons, au cours des précédents chapitres, indiqué le contexte de cette réalisation : nous voulons concevoir un support d'exécution, pour application parallèle irrégulières, sur machines à mémoire distribuée, qui soit facilement portable sur un noyau de système « classique », et qui permette une expression particulière des applications, à l'aide des concepts de poly-algorithmes et de décomposition procédurale parallèle.

Nous avons introduit la nécessité d'employer conjointement un noyau de multiprogrammation légère et une bibliothèque de communication portable. Nous en avons présenté des exemples assez précis. Quels choix de conception nous reste il à faire ?

Premièrement, nous devons préciser le modèle de programmation que nous envisageons : quelles entités seront manipulées, quelles interactions seront permises entre ces entités, quels opérateurs seront fournis pour exprimer cela...

Deuxièmement, nous devons choisir un modèle d'exécution : comment sera fait le couplage entre le noyau de multiprogrammation et la bibliothèque de communication, quel contrôle sera possible lors de l'exécution...

Enfin, nous devons identifier les limites imposées par ces choix.

Le modèle de programmation a été esquissé lors de la présentation des poly-algorithmes. Nous voulons permettre une expression sous forme poly-algorithmique, ce qui nous conduit au concept de décomposition procédurale parallèle. Ce concept est récursif ; une procédure appelée lors d'une décomposition parallèle peut elle même se décomposer en plusieurs sous- procédures parallèles. Nous voulons d'autre part recouvrir les temps d'attente de communication par des calculs utiles. Nous allons donc implanter chaque exécution de procédure à travers un mécanisme de RPC qui génère un nouveau fil d'exécution. Ce fil pourra se bloquer en attente de résultats partiels, et permettre automatiquement le recouvrement de cette attente par l'exécution d'une autre procédure.

Nous devons donc concevoir des opérateurs pour :

- rendre accessible la procédure à un appel à distance,
- exprimer le passage d'arguments (en tenant compte de l'hétérogénéité éventuelle),
- exprimer l'appel, l'attente de résultats.

Ces opérateurs sont somme toute assez classiques dans le monde client-serveur.

Nous nous restreindrons, dans un premier temps, à un modèle d'interaction uniquement client-serveur, pour contraindre le programmeur à suivre une expression procédurale, et vérifier l'adéquation même de cette expression pour les problèmes visés. Nous limiterons les possibilités de filtrage des appels à un simple mécanisme d'exclusion entre plusieurs appels d'un même point d'entrée. Nous devons encore préciser le contrôle de flux effectué lors de l'interaction client-serveur. Nous voulons présenter au programmeur une interface fiable, mais nous laissons le contrôle du flux des appels à l'application, à la fois à cause d'un problème lié à la bibliothèque de communication [1], mais aussi parce que ce contrôle doit être normalement pris en charge par la régulation de charge.

Nous envisageons un modèle dans lequel les procédures peuvent être aussi bien des fonctions pures (qui ne font que produire un résultat, et n'influent en rien sur le reste de leur environnement),

1. Nous en parlons en 92

que procéder à des effets de bords. En effet, le respect de la localité d'accès aux données nous semble nécessiter une identification des données indépendamment des actions qui leur sont appliquées. Nous choisissons donc de regrouper les données à l'aide d'une notion d'espace mémoire indivisible - la *tâche* - et de munir cet espace de toutes les procédures nécessaires à la manipulation de ces données - les *points d'entrée*. Nous oeuvrons dans le domaine du parallélisme, l'identification de ces entités sera donc explicite et ne passera pas par un mécanisme de localisation indirecte.

Pour permettre une gestion efficace des données, nous autoriserons le partage de la mémoire locale entre les différents fils d'exécution d'une même tâche. Pour assurer la cohérence, nous exploiterons les mécanismes de synchronisation usuels.

Nous souhaitons exprimer plusieurs modèles de calcul. Nous laisserons donc la possibilité de créer une nouvelle tâche, à n'importe quel moment et par n'importe qui. Nous ne contraindrons pas non plus les interactions entre les tâches à suivre un schéma particulier. Nous nous efforcerons cependant à assurer une procédure de démarrage et de terminaison correcte des composants d'un programme parallèle. En particulier, nous ne laisserons pas le programmeur avec une application partiellement interrompue en cas d'erreur.

Nous permettrons l'identification des caractéristiques des machines physiques et la construction dynamique d'une machine virtuelle. Nous nous baserons pour cela, autant que possible, sur les mécanismes déjà disponible au niveau du système d'exploitation ou de la bibliothèque de communication.

Nous informerons partiellement le programmeur de l'état de charge de la machine virtuelle, de façon à permettre une meilleure implantation de la couche de régulation de charge. Nous n'offrirons comme rétroaction que les possibilités d'évolution des machines physique et virtuelle et la régulation des appels de procédures. De plus, un certain contrôle de l'ordonnancement des actions sera possible à l'aide des primitives de synchronisation. Par contre, nous excluons dans un premier temps la possibilité de migration de processus. Nous considérons en effet que le grain des applications sera suffisamment fin pour que seul le placement soit nécessaire pour effectuer une bonne régulation de charge.

Nous nous baserons sur les mécanismes systèmes disponibles pour réaliser le stockage temporaire et permanent de données, la mesure du temps, l'interaction avec l'utilisateur.

Nous voulons introduire la multiprogrammation légère pour permettre un recouvrement automatique des attentes de communication et pour autoriser une expression simple d'une multiplicité de séquences indépendantes. Nous ne voulons pas compliquer la vie du programmeur d'application à cause de cette multiprogrammation. Nous choisissons donc de limiter les problèmes induits par la concurrence en n'autorisant pas d'exécution préemptive. Le programmeur d'application exécute chacune de ses séquences en exclusion. Il n'a donc généralement pas besoin de verrouiller ses données pour leur assurer un accès cohérent. Il peut aussi utiliser n'importe quelle bibliothèque même non fil-sauve, sans synchronisation supplémentaire. L'auto-ordonnancement des fils d'exécution ne sera pas directement contrôlable : c'est le noyau de multiprogrammation qui effectuera cette fonction. L'introduction de priorités n'est pas, dans un premier temps, exploré. La couche de régulation de charge agira en amont, en régulant les actions possibles. Le déroulement de l'exécution sera automatique, entièrement sous le contrôle du noyau de multiprogrammation.

Nous ne voulons pas redéfinir une nouvelle bibliothèque de communication, intégrant la multiprogrammation. Nous voulons apporter le minimum de modifications à une bibliothèque existante pour permettre son utilisation conjointe avec un noyau de multiprogrammation. Dans la mesure où, avec la bibliothèque de communication choisie, les communications ne peuvent pas interrompre les calculs en cours, nous serons amenés à définir une opération couplant progression des calculs et progression des communications. Différentes approches peuvent être envisagées pour ce couplage :

- la scrutation du réseau est effectuée lors d'actions explicites du programmeur,

- la scrutation est effectuée implicitement lors de certaines actions, comme les attentes de communication,
- la scrutation est effectuée lors de certaines étapes d'ordonnancement,
- la scrutation est effectuée lorsqu'aucun fil d'exécution n'est prêt à tourner,
- la scrutation est effectuée à l'aide d'une interruption périodique.

Nous avons choisi d'implanter les quatre premières propositions. En effet, nous offrons une primitive pour forcer la scrutation et nous effectuons cette action implicitement à chaque attente. D'autre part nous effectuons automatiquement une scrutation à chaque ré- ordonnancement, ou bien lorsqu'aucun fil d'exécution n'est prêt à tourner. Par contre, nous n'offrons pas de scrutation grâce à une interruption périodique, puisque les applications semblent se contenter de la proposition actuelle. Cependant, cette dernière approche est relativement facile à implanter, puisqu'il suffit de rendre PVM *signal-safe*, par exemple en interdisant l'exécution du traitant d'interruption périodique lorsqu'une primitive de PVM est en cours d'exécution.

Trois grandes limitations sont introduites par ces choix.

De part son fonctionnement en coroutinage, Athapascan-0a ne permet pas un parallélisme vrai entre les processus légers d'une même tâche. Il ne peut donc pas tirer efficacement parti d'un multiprocesseur symétrique. Notons toutefois que l'absence de préemption n'est pas inhérente au modèle de programmation choisi ; une implantation préemptive de ce modèle permettrait donc une exécution efficace sur une machine SMP.

L'absence de préemption implique aussi l'absence de partage de temps entre les fils d'exécution. Le temps partagé sert principalement à garantir l'exécution « de front » d'un ensemble de travaux. Ce mécanisme est intéressant associé à des priorités, pour diriger l'exécution des processus légers, par exemple dans une application combinatoire pour laquelle on dispose d'une heuristique. Cependant, la cible principale d'Athapascan-0 étant le calcul « pur » et l'ordonnancement des calculs étant du ressort d'Athapascan-1, nous avons décidé de ne pas utiliser la préemption.

Enfin, une troisième limitation est liée à l'absence de scrutation périodique, ce qui peut dans certains cas compliquer l'implantation d'utilitaires de régulation de charge, qui nécessite souvent un échange périodique d'états de charge. Cependant, ce mécanisme est relativement simple à mettre en œuvre. Ce sera donc une évolution du prototype, qui interviendra lorsque les couches de régulation de charge seront suffisamment avancées.

4.3 L'organisation d'Athapascan-0a

La couche Athapascan-0a peut se décomposer en deux blocs distincts :

- une *interface de programmation* : basée sur le langage C [Kernighan & Ritchie 1978], elle y ajoute quelques macro-instructions, et un ensemble de primitives ; par abus, nous l'appellerons couramment le « langage » Athapascan-0a. Cette interface de programmation est la partie visible de la couche Athapascan-0a, pour les autres membres du projet.
- un *exécutif* : il regroupe tous les mécanismes permettant à la couche Athapascan-0 de fonctionner. Nous l'appellerons couramment "noyau" car il encapsule ces mécanismes. Il est portable. Pour assurer sa portabilité, le noyau Athapascan-0a se base sur deux sous-composantes, un noyau de multiprogrammation et un noyau de communication.

Nous allons décrire ces deux composantes, mais d'abord nous devons présenter plus en détail le modèle de programmation d'Athapascan-0a.

4.4 Le modèle de programmation d'Athapascan-0a

Le modèle de programmation est caractérisé par le mécanisme d'interaction entre séquence qui est employé, les différents schémas d'organisation des calculs qui peuvent être développés, la gestion de la multiprogrammation et enfin un ensemble de fonctionnalités annexes de contrôle de l'environnement ambiant et d'information. Nous présentons tour à tour chacun de ces points, mais nous allons d'abord définir notre terminologie dans le cadre du système de programmation Athapascan-0.

4.4.1 Terminologie utilisée

Nous définissons ici les différentes entités qui composent le modèle de programmation d'Athapascan-0a.

- **Nœud** (*node*) : Matériel qui supporte les fonctions de calcul et de communication. Un nœud peut comporter plusieurs processeurs.
- **Tâche** (*task*) : Processeur Virtuel qui est placé sur un nœud. Chaque tâche est indépendante des autres et de sa localisation. Toutes les tâches sont passives, elles ne font que réagir aux requêtes qui leurs sont posées. Un mécanisme d'initialisation / terminaison de tâche doit être fourni.
- **Point d'entrée** ou **Service** (*entry point* ou *service*) : Service offert par une tâche, invoqué par une requête. Une tâche gère elle-même ses points d'entrée. Un point d'entrée peut être *utilisateur* s'il est défini par l'utilisateur ou *système* s'il est défini par l'exécutif Athapascan.
- **Modèle de point d'entrée** (*entry point model* ou *service function*) : c'est la fonction (dite « de service ») qui sera effectivement exécutée sur réception d'une requête pour le point d'entrée considéré.
- **Fil d'exécution** (*thread*) : Processus léger qui traite une requête sur un service. Un fil d'exécution active la tâche appelée, ou plus exactement la fonction de service associée au service demandé.
- **Requête / Réponse** (*request / response*) : Messages échangés durant le mécanisme d'invocation bloquante ou non-bloquante d'un service. Toutes les requêtes demandent une réponse.
- **Point de synchronisation** (*synchronization point*) : fait le lien entre une réponse reçue et la requête correspondante envoyée de façon non-bloquante à un service. Le point de synchronisation est un objet opaque maintenu par l'exécutif.
- **Modèle de tâche** (*task model*) : Description de l'implantation des différents modèles de points d'entrée que peut offrir une tâche. Un modèle de tâche abstrait (source) peut se traduire en plusieurs modèles de tâche physiques (binaires). Une tâche est en fait une instance d'un modèle de tâche (physique).
- **Programme** (*program*) : Application parallèle. Un programme consiste en un ensemble de modèles de tâches, en une description de nœuds (sur lesquels s'exécutera l'application) et une section programme.
- **Section programme** : Ensemble d'instructions destinées à installer les tâches sur les nœuds, initialiser ces tâches, lancer le calcul, récupérer le résultat, et terminer les tâches. Cet ensemble d'instructions est exécuté par une tâche dite **tâche racine du programme**.

4.4.2 Mécanisme d'interaction

Le mécanisme d'interaction d'Athapascan est l'appel de procédure à distance (RPC). Contrairement aux implantations courantes du RPC [Birrell & Nelson 1984], celui d'Athapascan est asynchrone. Ce mécanisme permet d'exprimer simplement la décomposition procédurale parallèle. Un

programme Athapascan est un formé d'un ensemble de *tâches* de calcul. Chaque tâche exporte un certain nombre de *points d'entrée*, c'est à dire les déclare comme accessibles depuis d'autres tâches. Un point d'entrée est analogue à une procédure, mais peut être appelé à distance par une autre tâche. L'instanciation d'un point d'entrée est suivie d'une attente de résultats. Ces deux opérations sont disjointes, l'utilisateur récupérant un *point de synchronisation* à l'issue de l'instanciation et utilisant ce point de synchronisation pour attendre les résultats retournés par la tâche distante. Il est ainsi possible d'appeler plusieurs tâches en parallèle, sur divers points d'entrée, en effectuant plusieurs instanciations et plusieurs attentes de résultats.

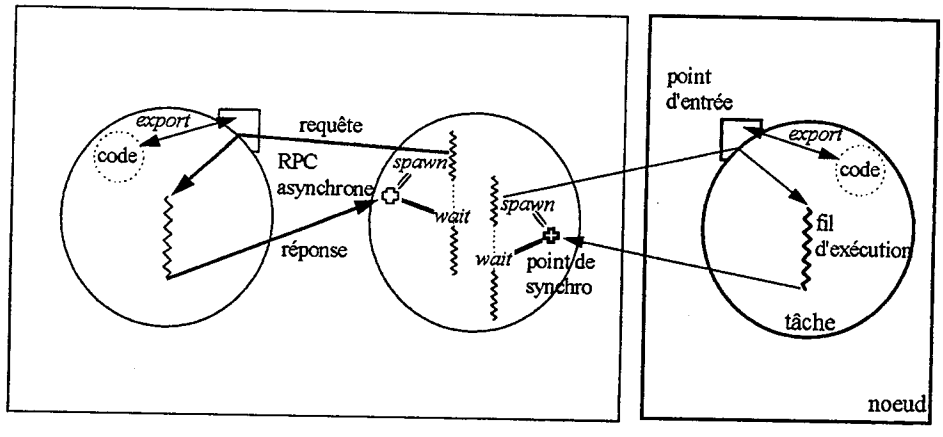

FIG. 4.1 - *Le mode d'interaction d'Athapascan-0a.*

Les fonctions de service sont exportées. Un fil d'exécution peut effectuer un RPC asynchrone sur un service distant (spawn), puis attendre la complétion de ce service (wait sur un point de synchronisation établi lors du spawn). L'activation d'un service correspond à la création d'un nouveau fil d'exécution. La multiprogrammation permet de recouvrir les attentes de communication par du calcul.

Les paramètres et les résultats sont emballés et déballés depuis des *tampons de communication* et transmis comme composante des messages d'instanciation et de résultat. Un point d'entrée d'une tâche est donc simplement une *fonction* prenant en entrée un tampon de communication contenant les paramètres d'appel et retournant un nouveau tampon de communication contenant les résultats de l'appel. Le niveau Athapascan-0 est un niveau de base : l'utilisateur doit écrire ses propres talons, les routines qui font l'emballage et le déballage des paramètres / résultats dans les tampons de communication, lors de chaque appel d'un service. Cependant, cette tâche pourrait être effectuée par un compilateur (comme pour IDL [Otte et al. 1996] ou RPC-GEN [Bloomer 1992, Gibbons 1987]), ou bien encore par une couche « interface utilisateur » qui proposerait des concepts d'abstraction de données, réalisée par exemple en C++ [Stroustrup 1991]. Une telle extension est réalisée par la bibliothèque PAC++ [Gautier et al. 1994] qui emploie une couche d'interface en C++ pour écrire ses talons Athapascan. En Athapascan-0a, l'emballage et le déballage des paramètres dans les tampons de communication se fait d'une façon très semblable à celle de PVM : l'utilisateur emballe / déballe successivement toutes les données, spécifiant leur type, leur adresse de base et éventuellement leur étendue s'il s'agit de données contiguës (tableaux). Les types de base supportés sont les même que ceux de PVM.

Un opérateur d'enregistrement d'un point d'entrée est fourni, permettant d'associer une fonction de service particulière à un numéro de point d'entrée particulier. Cette association n'a qu'un sens local à la tâche où elle est réalisée. C'est au programmeur de garder une cohérence suffisante dans sa numérotation des points d'entrée des diverses tâches. Typiquement, les utilisateurs d'Athapascan définissent de façon globale tous les numéros des points d'entrée de leur programme. Cette définition

globale peut compliquer la compilation séparée ou la distribution de bibliothèques en Athapascan, puisqu'il faut assurer l'unicité de chaque numéro de point d'entrée. Cependant, c'est en quelque sorte une phase de liaison « manuelle » du programme, qui pourrait être automatisée par un outil adéquat.

Le choix du site d'exécution d'un RPC est à la charge du programmeur ou d'un outil de régulation de charge. Athapascan-0a n'offre que le support d'activités explicitement placées.

4.4.3 Multiprogrammation

Lorsqu'une tâche effectue un ou plusieurs appel(s) de point d'entrée, puis attend les résultats de l'un d'eux, la tâche n'est pas laissée inactive. Athapascan utilise la multiprogrammation légère pour recouvrir les temps d'attente de résultat par des calculs. Pour cela, chaque instanciation d'un point d'entrée est implanté comme l'activation d'un nouveau fil d'exécution. L'utilisation de processus légers permet de réduire le grain de parallélisme exploitable, en masquant les attentes de communication par d'autres calculs.

Lors de la définition d'un point d'entrée, il est possible de spécifier une limite à son exécution concurrente ; typiquement, cela permet d'assurer une exécution atomique du service. Cependant, Athapascan n'offre pas de mécanisme évolué de *garde* des points d'entrée. Un fonctionnement similaire peut être réalisé à l'aides de variables de conditions manipulées par les processus légers. Les requêtes originaires d'une même tâche et pour un même point d'entrée d'une tâche donnée sont traitées dans leur ordre d'émission (PVM garantissant un ordre des messages FIFO entre les processeurs), mais les requêtes originaires de tâches différentes, à destination de tâches différentes, ou bien pour des points d'entrée différents peuvent être traitées dans n'importe quel ordre. Toute possibilité de famine dépend du noyau de communication employé. Comme nous le verrons dans la suite, Athapascan-0a n'offre aucun contrôle de flux ; cependant le nombre de requêtes pendantes doit être borné. Cela est du ressort de l'utilisateur ou d'une couche de régulation de charge.

L'utilisateur peut, lors de l'appel d'un service, inclure un *surnom* permettant par exemple de distinguer les différentes activations d'un même point d'entrée. Ce surnom est un simple entier.

Chaque fil exécutant une requête sur un point d'entrée possède :

- une mémoire privée (sa pile et son tas),
- une mémoire commune avec les autres fils exécutant le même point d'entrée (la mémoire de point d'entrée),
- une mémoire commune avec tous les autres fils d'exécution de la tâche,
- une mémoire commune avec tous les fils d'exécution de toutes les tâches du programme (« l'environnement »).

L'accès aux différentes mémoires n'est pas identique. La pile et le tas du fil d'exécution sont accessibles directement, par des variables dynamiques de C. La mémoire commune de la tâche est accessible par l'intermédiaire de variables statiques de C. La mémoire commune à tous les fils exécutant le même point d'entrée est accessible par une primitive spéciale. Enfin l'environnement est accessible par des fonctions particulières, similaires à celles que l'on trouve dans C pour l'accès à l'environnement d'un programme. Notons qu'il n'y a pas de mémoire commune inter-tâches dans le sens « mémoire virtuelle distribuée ».

Il n'y a pas de mécanisme de synchronisation *globale* (inter-tâches) pour accéder aux mémoires communes. Il est cependant possible de réaliser l'exclusion mutuelle entre les fils d'exécution d'une même tâche, à l'aide de sémaphores.

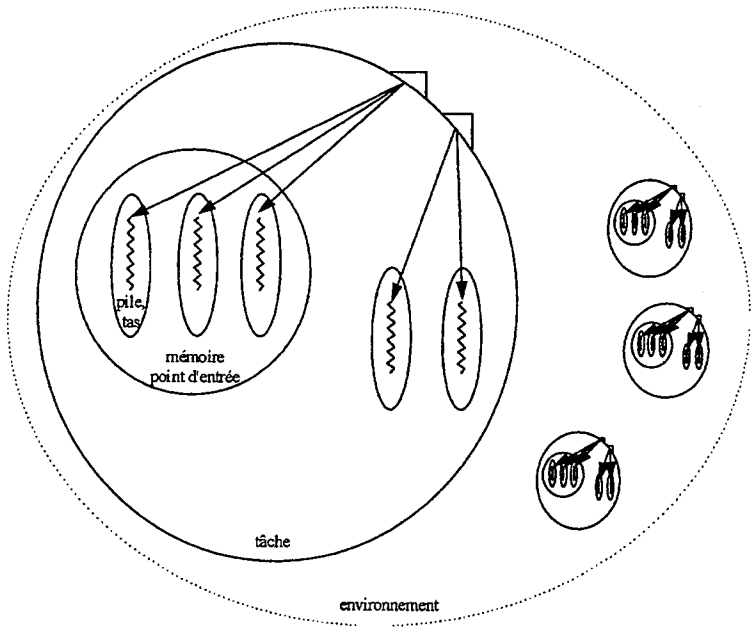

FIG. 4.2 - *L'organisation mémoire dans Athapascan-0a.*

Les différentes catégories de mémoire sont illustrées ici : mémoire privée à un fil d'exécution, mémoire de point d'entrée, mémoire partagée de la tâche et environnement commun inter-tâches.

4.4.4 Modèle de calcul

Un programme Athapascan est exécuté dans un mode client-serveur généralisé ; toute tâche peut être à la fois *cliente* (appeler les points d'entrée d'autres tâches) et *serveur* (déclarer et servir des points d'entrée). Une première tâche, *racine*, est lancée. Elle peut démarrer d'autres tâches sur d'autres processeurs, qui peuvent à leur tour démarrer de nouvelles tâches. Une tâche peut décider de se suicider. Le comportement dynamique des tâches en Athapascan est à la charge de l'utilisateur : c'est à lui de garder en mémoire quelles tâches sont actives à un moment donné. Bien que cela impose au programmeur le souci de gérer lui-même ses tâches, cela lui permet aussi la plus grande souplesse dans leur gestion. La tâche Athapascan-0a est en fait une tâche PVM et est seulement un conteneur pour exécuter des points d'entrée. L'identificateur d'une tâche Athapascan-0a est simplement l'identificateur fourni par PVM ; il est donc global et unique. La tâche racine effectue l'agrégation de processeurs physiques dans la machine parallèle virtuelle, la création de nouvelles tâches, leur initialisation, éventuellement leur activation puis leur terminaison, enfin la libération des processeurs physiques. La tâche racine peut donc être employée pour suivre un modèle maître-esclave. Cependant ce modèle n'est pas le seul possible ; nous avons vu qu'une tâche fille peut créer de nouvelles tâches ; il est donc envisageable de déployer un arbre de tâches sur plusieurs niveaux ; il est possible de spécialiser certaines tâches ou groupes de tâches ; enfin la tâche racine peut simplement se « dupliquer » pour aboutir à un modèle de type SPMD.

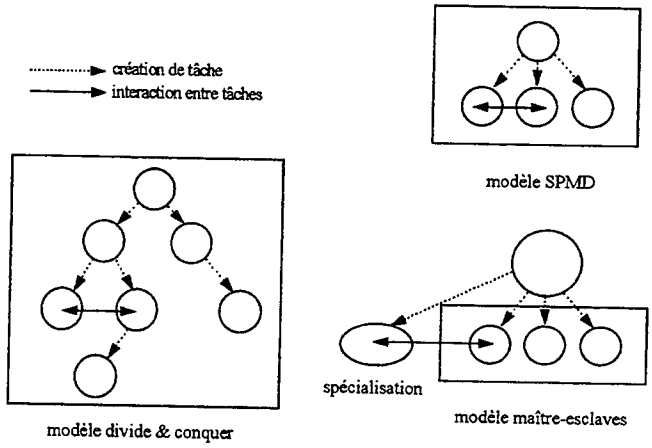

FIG. 4.3 - *Les modèles de calcul d'Athapascan-0a.*

Différents modèles de calcul. L'interaction entre tâches est indépendante de l'établissement des tâches.

Notons que l'identification d'une tâche n'est connue originellement que par sa tâche créatrice. Il est donc nécessaire, pour autoriser l'envoi de requêtes entre deux tâches quelconques, de communiquer l'identification de la tâche qui servira de serveur à la tâche qui en sera cliente. Cette gestion des identificateurs de tâches est à la charge du programmeur mais est généralement simple à effectuer : par exemple une tâche peut créer un ensemble de tâches filles, puis les initialiser en leur passant comme paramètre un vecteur contenant tous les identificateurs des autres tâches créées. L'activation des tâches filles n'est pas forcément le fait de leur tâche parente ; il est envisageable que la tâche parente ne serve qu'à créer et initialiser un ensemble de tâches filles qui se connaissent entre-elles. Ces tâches filles peuvent alors s'appeler les unes les autres sans intervention de leur parent. De même, la terminaison des tâches n'est pas forcément à la charge de la tâche racine ; cependant, considérer la tâche racine comme un agrégateur de résultat final est la façon la plus simple de terminer un programme Athapascan. Lorsque le résultat final est complet, la tâche racine peut alors terminer toutes les tâches, y compris elle-même. D'autres modèles sont possibles, par exemple où la terminaison est décidée de façon distribuée. Athapascan ne fournit pas d'opérateur ad-hoc pour détecter la *quiescence* d'un calcul, cependant, tout algorithme de terminaison distribuée peut être envisagé, son implantation étant à la charge du programmeur.

Lorsqu'une tâche fille est créée, elle est passive (une tâche fille n'a pas de section programme). Il faut donc l'initialiser pour qu'elle puisse déclarer les points d'entrée qu'elle doit servir. Cette initialisation se fait à travers un point d'entrée spécial, d'initialisation. De même, la terminaison d'une tâche se fait à travers un point d'entrée spécial, de terminaison. Ces deux points d'entrée sont obligatoires pour toute tâche. Le point d'entrée d'initialisation d'une tâche fille doit être appelé précédemment à tout autre : son code, qui doit être défini par le programmeur pour chaque tâche, doit « initialiser » la tâche : en particulier, déclarer tous les autres points d'entrée que va servir la tâche. Typiquement, ce point d'entrée est aussi employé pour installer des données dans la tâche. Le point d'entrée de terminaison est le dernier point d'entrée qui doit être appelé sur la tâche. Son code est aussi spécifié par le programmeur, mais peut être laissé vide. Ce point d'entrée permet par exemple de rapatrier les résultats finaux, conservés auparavant dans la tâche et de commander le suicide de la tâche. Ces deux points d'entrée encapsulent les appels aux points d'entrée utilisateur (voir fig. 4.4). Ils permettent de vérifier un bon déroulement de l'utilisation de la tâche. Des points d'entrée système sont aussi fournis par les tâches : par exemple, il existe un service de terminaison abrupte du programme parallèle (toutes les tâches sont terminées en catastrophe) et un service de terminaison abrupte d'une tâche donnée.

Notons qu'il est possible d'enregistrer de nouveaux points d'entrée (utilisateur) à tout moment (entre l'initialisation et la terminaison) de l'exécution d'une tâche ; de même, il est possible de fermer un point d'entrée à tout moment. Ces deux actions sont purement locales à la tâche considérée. Cependant, la cohérence entre les requêtes émises et les points d'entrée ouverts doit être assurée par le programmeur : si une tâche reçoit une requête pour un point d'entrée qui n'est pas enregistré, c'est une erreur d'exécution (et de programmation), fatale.

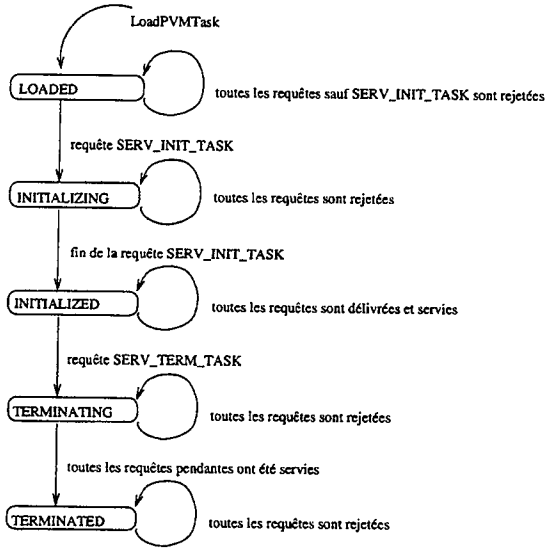

FIG. 4.4 - *Le cycle de vie d'une tâche.*

Une tâche démarrée par la primitive LoadPVMTask doit être initialisée par une requête SERV_INIT_TASK pour le service d'initialisation. Elle sert ensuite toutes les requêtes utilisateur et système, jusqu'à ce qu'une requête SERV_TERM_TASK soit reçue.

4.4.5 Contrôle et information

Pour faciliter la programmation en Athapascan, les tâches partagent une sorte d'environnement global : bien que d'accès lent, cet environnement peut être employé pour faire connaître des informations à toutes les tâches d'un programme. Par exemple, il est possible de définir une tâche *console* chargée des interactions avec l'utilisateur (console en mode texte ou bien console de visualisation graphique). L'identification de cette tâche console peut alors être transmise à toutes les tâches par l'intermédiaire de l'environnement global. Notons que cet environnement global n'est pas une mémoire partagée d'usage « général » pour laquelle un fort débit ou un contrôle de cohérence seraient nécessaires.

Comme dans PVM, il est possible de connaître les différents processeurs de la machine parallèle virtuelle, d'en ajouter ou d'en retirer, à tout moment. Ici encore, le programmeur doit gérer la cohérence de la machine parallèle virtuelle, de la même façon qu'avec PVM.

Une opération d'ordonnancement-scrutation du réseau, permettant de faire progresser calculs et communications, est automatiquement réalisée sur certaines actions - comme l'attente d'un résultat. Elle peut aussi être forcée par le programmeur.

Enfin, Athapascan-0a permet la consultation de la charge locale engendrée dans une tâche. Cette information permet éventuellement à la couche Athapascan-1 de planifier les sites d'exécution des différents appels de procédures. Notons que la politique d'information de charge n'est pas du ressort d'Athapascan-0a ; seule l'extraction et la mise à disposition d'un indice de charge locale est réalisée.

4.4.6 Conclusion sur le modèle de programmation

Athapascan-0a se fonde sur le mécanisme de RPC asynchrone entre tâches de calcul. Des tampons de communication permettent d'emballer / déballer des arguments de façon indépendante du format des données. La multiprogrammation légère permet de recouvrir les attentes de résultats. Le modèle de calcul, le client-serveur généralisé, est très lâche et permet plusieurs formes de chargement et d'activation de tâches. Un cycle de vie des tâches doit cependant être respecté. Des informations sur la machine physique et la machine virtuelle peuvent être obtenues. Un environnement global et des tâches consoles complètent l'exécutif.

4.5 L'interface de programmation d'Athapascan-0a

Nous décrivons dans la suite l'interface de programmation qui a été définie pour la couche Athapascan-0a. Cette interface de programmation permet la spécification d'applications suivant le modèle précédemment décrit. Cette interface de programmation étend le langage C et est appelée abusivement le « langage » Athapascan-0a. L'interface peut être découpée schématiquement en trois parties :

- la spécification des RPC (la programmation d'un service, son exportation, l'appel synchrone et asynchrone, la transmission de données comme paramètres ou résultats) ;
- la gestion des machines physique et virtuelle (agrégation de processeurs physiques, démarrage de processeurs virtuels, informations...)
- la gestion du partage de mémoire locale (gestion mémoire et synchronisation locale).

4.5.1 Squelette d'une tâche Athapascan-0a

Dans un fichier source en langage C, on définit un modèle de tâche Athapascan-0a de la façon suivante :

```
#include "Ath0/ath0.h"
TASK_MODEL_MACRO (nom_du_modèle_de_tâche, argc, argv)
définition des variables globales à tous les fils d'exécution
définition des procédures "utilitaires"
déclaration des modèles de points d'entrée :
EP_MODEL_MACRO (fonction_de_service1, in, out)
BEGIN_EP
code du service
END_EP
EP_MODEL_MACRO (fonction_de_service2 ... )
...
EP_MODEL_MACRO (fonction_de_serviceN... )
...
EP_MODEL_MACRO (InitTask, in, out)
BEGIN_EP
initialisation des variables globales
déclaration des points d'entrée :
NewEntryPoint(...)
NewEntryPoint(...)
END_EP
EP_MODEL_MACRO (TermTask, in, out)
BEGIN_EP
il peut éventuellement ne pas y avoir de code ici.
END_EP
END_TASK
```

FIG. 4.5 - *Le squelette d'une tâche Athapascan-0a.*

Le fichier ath0.h contient les définitions du langage Athapascan-0a.

Une tâche doit obligatoirement offrir les deux fonctions de points d'entrée : InitTask (pour initialiser la tâche) et TermTask (pour terminer proprement la tâche). Elles sont respectivement associées par le noyau aux services standards SERV_INIT_TASK et SERV_TERM_TASK. SERV_INIT_TASK est le premier service appelé sur une tâche, et SERV_TERM_TASK est le dernier. Si TermTask peut éventuellement ne contenir aucun code, InitTask doit au moins déclarer les points d'entrées initiaux de la tâche.

4.5.2 Programmation d'un modèle de point d'entrée

Un modèle de point d'entrée est organisé typiquement de la façon suivante :

```
EP_MODEL_MACRO (fonction_de_service, in, out)
BEGIN_EP
déballage des arguments d'appel du service :
Upk...(in, ...) ; ...
...utilisation des arguments d'appel du service...
emballage des arguments de la réponse :
Pk...(out, ...) ; ...
la réponse est envoyée automatiquement à la fin de l'exécution du fil.
END_EP
```

FIG. 4.6 - *Un modèle de point d'entrée.*

Le descripteur des arguments d'appel, *in*, et le descripteur des arguments résultats, *out*, sont alloués et libérés automatiquement par le noyau.

Un fil qui exécute une fonction de service, peut connaître :

- quelle est la tâche qui l'appelle, par la primitive **GetCallerId**,
- dans quelle tâche il s'exécute, par la primitive **GetTaskId**,
- quelle identification lui a attribué le programmeur, par la primitive **GetNickName**.

4.5.3 Déclaration d'un point d'entrée

Une fois défini un modèle de point d'entrée, il faut l'associer à un point d'entrée par la primitive **NewEntryPoint**, pour qu'il puisse faire l'objet de requêtes provenant de l'extérieur. La primitive **NewEntryPoint** prend comme paramètres la fonction de service, un numéro d'identification du service, une limite de concurrence et un pointeur sur la zone de mémoire privée aux fils qui exécutent le service. Les services sont désignés par un numéro d'ordre. L'étendue des numéros de service va de 1 jusqu'à **ServiceMaximumNumber**. . La fonction de service associée à un service peut être redéfinie à tout moment par un nouveau **NewEntryPoint**, ou bien remplacée par un service qui répond « abonné absent », par la primitive **RemoveEntryPoint**. Certains services standards existent : SERV_INIT_TASK et SERV_TERM_TASK en particulier.

4.5.4 Appel d'un service : méthode primitive

Il existe deux formes d'appel d'un service : la méthode primitive, décrite ici, et une méthode plus évoluée, décrite plus loin. Ces deux formes sont équivalentes et peuvent être utilisées conjointement.

4.5.4.1 Forme générale de l'appel d'un service

Un appel d'un service est effectué par une séquence du type :

```
allocation d'espace pour les arguments d'appel, initialisation de l'appel :
AllocArg(... ,&arg) ;
emballage des arguments d'appel :
Pk...(arg, ...) ; ...
appel du service (synchrone ou asynchrone) :
Appel(arg, &res) ;
déballage des arguments résultats :
Upk...(res, ...) ; ...
libération de l'espace alloué aux arguments résultats :
FreeRes(res) ;
```

FIG. 4.7 - *Forme générale de l'appel d'un service.*

Le destinataire de la requête, le service appelé et le nom choisi par le programmeur pour l'instance du service qui sera créée, sont spécifiés lors de l'allocation d'espace pour les arguments d'appel.

Les arguments d'appel, décrits par *arg*, sont automatiquement libérés à la fin de l'appel et les arguments résultats, décrits par *res*, sont automatiquement alloués par le noyau.

L'appel (Appel) peut être effectué par **DoCall** ou **Dospawn / WaitSpawn** pour exprimer respectivement un appel synchrone ou asynchrone.

4.5.4.2 Appel synchrone

L'appel synchrone s'effectue comme suit :

```
DoCall(arg, &res) ;
```

FIG. 4.8 - *L'appel synchrone.*

Le fil qui exécute cet appel perd le processeur le temps de recevoir la réponse.

4.5.4.3 Appel asynchrone

L'appel asynchrone se fait en plusieurs étapes :

```
DoSpawn(arg, &sp) ;
...
test de terminaison du service appelé:
if (TestSpawn(sp)==ATL_TESTOK) ...
...
attente de terminaison du service appelé:
WaitSpawn(sp, &res)
```

FIG. 4.9 - *L'appel asynchrone.*

Le point de synchronisation, *sp*, alloué lors du **DoSpawn**, est automatiquement libéré par le **WaitSpawn**. **DoSpawn** ne fait pas perdre le processeur au fil qui l'exécute, mais **TestSpawn** et **WaitSpawn** le font.

4.5.4.4 Primitives d'emballage / déballage

Les primitives d'emballage suivantes ont été définies :

Pk{Short,Int,Long}	emballe un entier,
PkArray{Short,Int,Long}	emballe n éléments consécutifs d'un vecteur d'entiers,
Pk{Float,Double}	emballe un réel,
PkArray{Float,Double}	emballe n éléments consécutifs d'un vecteur de réels,
PkTaskId	emballe un identificateur de tâche,
PkByte	emballe une suite d'octets,
PkStr	emballe une chaîne 0-terminée.

TAB. 4.1 - *Primitives d'emballage.*

Les primitives **Upk**... correspondantes permettent le déballage des résultats.

4.5.5 Appel d'un service : méthode « en une ligne »

En plus de la méthode primitive de l'appel d'un service, il existe une méthode évoluée dont le but est simplement d'être un sucre syntaxique permettant d'améliorer la lisibilité du source en offrant "un appel en une ligne". Bien que plus simple d'emploi, cette méthode est moins générale que la méthode primitive.

4.5.5.1 Les primitives Pack et UnPack

Les primitives **Pack** (**UnPack**) correspondent à une série de **Pk.. (Upk..)** :

```
Pack(..., fmtarg, arg1, .., argn);
UnPack(..., fmtres, res1, .., resn);
```

FIG. 4.10 - *Les primitives Pack et Unpack « en une ligne ».*

Les formats *fmtarg* et *fmtres* précisent le nombre et les types des arguments à emballer et déballer (à la mode de *printf*).

Les types des arguments que l'on peut spécifier sont :

- l'entier (h ou d ou l),
- le vecteur dimensionné d'entiers (H ou D ou L),
- le réel (f ou g),
- le vecteur dimensionné de réels (F ou G),
- l'identificateur de tâche (t),
- le vecteur dimensionné d'identificateurs de tâches (T),
- le vecteur dimensionné d'octets (B) et la chaîne de caractères (s).

Un format est une chaîne de caractères contenant des lettres parmi "hdlHDLfgFGtTsB", éventuellement précédées par un nombre positif.

Exemple :

"d3D" signifie un entier suivi de trois vecteurs dimensionnés d'entiers. Un vecteur dimensionné est un pointeur sur le premier élément du vecteur précédé du nombre d'éléments, dans la liste des paramètres.

4.5.5.2 Appel synchrone « en une ligne »

La primitive **Call** permet de faire en une ligne :

- l'initialisation de l'appel,
- l'emballage des arguments,
- l'appel proprement dit (perte du processeur),
- le déballage des résultats,
- la terminaison de l'appel.

Pour cela, la primitive **Call** admet un nombre variable d'arguments et de résultats :

```
Call(..., fmtarg, arg1, .., argn, fmtres, res1, .., resn);
```

FIG. 4.11 - *L'appel synchrone « en une ligne ».*

Un format NULL signifie aucun arguments à emballer ou à déballer. La primitive **Call** en une ligne est analogue à une séquence **AllocArg, Pack, DoCall, UnPack, FreeRes**.

4.5.5.3 Appel asynchrone « en une ligne »

De même que pour le **Call**, une forme évoluée pour l'appel asynchrone est disponible :

```
Spawn(..., fmtarg, arg1, .., argn, &sp);
...
test de terminaison du service appelé:
if (TestSpawn(sp)==ATL_TESTOK) ...
...
attente de terminaison du service appelé:
WaitSpawnRes(sp, fmtres, res1, .., resn)
```

FIG. 4.12 - *L'appel asynchrone « en une ligne ».*

Le point de synchronisation, *sp*, alloué lors du **Spawn**, est automatiquement libéré par le **WaitSpawnRes**. **Spawn** ne fait pas perdre le processeur au fil qui l'exécute, mais **TestSpawn** et **WaitSpawnRes** le font.

4.5.6 Squelette d'un programme Athapascan-0a

Un *programme* Athapascan-0a (la tâche racine) est un fichier source en langage C, du type :

```
#include "ath0.h"
PROGRAM_MACRO (nom_du_programme, argc, argv)
définition des variables globales à tous les fils d'exécution
définition des procédures "utilitaires"
définition de modèles de point d'entrée
BEGIN_PROGRAM
code du programme
END_PROGRAM
```

FIG. 4.13 - *Le squelette d'un programme Athapascan-0a.*

Le code du programme doit :

- instancier des points d'entrée d'après les modèles que la tâche possède,
- définir les nœuds utilisés par la Machine Virtuelle Athapascan, par les primitives **NewProc** et **RemoveProc**, ou bien les reconnaître, par la primitive **GetProcs**,
- charger les instances des modèles physiques des tâches sur les nœuds, par la primitive **LoadPVMTask**,
- initialiser toutes les tâches par leur service SERV_INIT_TASK,
- éventuellement, effectuer des calculs en appelant un ou plusieurs services proposés par les tâches,
- si besoin, terminer proprement les tâches par leur service SERV_TERM_TASK.

Cette séquence ne doit pas obligatoirement se trouver dans la construction **PROGRAM**, certaines tâches pouvant se charger de gérer des "sous-tâches". Mais la construction **PROGRAM** devrait au moins établir une tâche pour pouvoir effectuer des invocations à ses services.

4.5.7 Chargement et Terminaison d'une tâche

Le chargement d'une tâche est effectué par la primitive **LoadPVMTask**. Cette primitive prends comme argument le nom du processeur physique où la tâche doit être créée et le nom du binaire à charger. Elle retourne une identification de la tâche créée. Ce mécanisme est dérivé de celui de PVM, d'où son nom.

La terminaison d'une tâche peut se faire de différentes façons. Le service SERV_KILL_TASK permet de terminer une tâche. Le service SERV_EXIT_PROG permet de terminer toutes les tâches d'un programme. Ces deux services sont de bas niveau. Alternativement, la primitive **AthExit** permet de terminer le programme parallèle, en renseignant l'utilisateur sur la cause de la terminaison par une chaîne de caractères. Toutes ces terminaisons sont abruptes (des requêtes en cours peuvent ne pas être exécutées). Il n'y a pas de mécanisme de détection de quiescence d'un calcul.

4.5.8 Désignation d'une tâche

Les tâches sont désignées par un identificateur géré par le système. Les identificateurs de tâches peuvent être communiqués à l'aide des primitives **PkTaskId** et **UpkTaskId**. Ils peuvent aussi être converti en / depuis une chaîne par les primitives **TaskIdToChar** et **CharToTaskId**. Chaque programme Athapascan-0a est doté d'un environnement qui permet de communiquer des identificateurs de tâches (par exemple l'identificateur de la tâche console).

4.5.9 Primitives d'information

La primitive **GetProcs** permet de connaître les processeurs physiques participant à la machine virtuelle.

Les primitives **GetTotalThreads** et **GetReadyThreads** permettent de connaître le nombre total de fils d'exécution et le nombre de prêts, respectivement, dans la tâche courante.

La primitive **GetTaskId** permet de connaître l'identité de la tâche courante.

La primitive **GetCallerId** permet de connaître l'identité de la tâche qui a appelée l'instance de service courante.

La primitive **GetNickname** permet de connaître le surnom donné à l'instance de service courante.

La primitive **GetInfoEP** permet d'accéder à la zone mémoire privée à tous les fils d'exécution d'un service donné.

4.5.10 Primitives de gestion mémoire

Les primitives **LocalMalloc**, **LocalFree**, **GlobalMalloc**, **GlobalFree** permettent d'allouer / libérer des blocs de mémoire, privée pour le fil d'exécution, ou bien globalement dans la tâche. En fait, toute mémoire dynamique est globale à la tâche ; cependant, le programmeur peut la restreindre à un fil d'exécution particulier en ne donnant pas l'adresse du bloc mémoire aux autres.

4.5.11 Primitives de synchronisation et d'ordonnancement

Des primitives sont fournies pour synchroniser les fils d'exécution d'une même tâche (synchronisation intra-tâche). Les primitives **NewLSem, LSemP, LSemV** implantent des sémaphores et les

primitives **NewEvent, WaitEvent, PostEvent** implantent des compteurs d'événements [Reed & Kanodia 1979]. Ces primitives permettent de synchroniser plusieurs fils d'exécution lors des accès à la mémoire commune locale. Il n'y a pas de primitives de synchronisation inter-tâches.

La primitive **Yield** permet de forcer une commutation vers un autre fil d'exécution prêt. La primitive **YieldnPoll** permet de forcer la progression des calculs et des communications en exécutant l'algorithme d'ordonnancement-scrutation décrit plus loin.

4.5.12 L'environnement des tâches

Toutes les tâches participant au même programme partagent un environnement analogue à l'environnement fourni par un shell sous Unix. Les primitives **PutEnv** et **GetEnv** permettent d'écrire et de lire dans l'environnement. Cet environnement peut servir en particulier à faire connaître l'identité de la tâche « console » qui peut être associée au programme.

4.5.13 Un exemple

L'interface de programmation d'Athapascan-0a cache donc complètement PVM. Elle définit un parallélisme de type client-serveur, entre tâches, qui est la base de la décomposition procédurale parallèle et de l'expression sous forme de poly-algorithmes imbriqués.

Pour prendre un exemple simple, regardons un produit scalaire de deux vecteurs. A première vue, l'algorithme consistant à calculer des produits scalaires partiels en parallèle, puis à sommer les résultats partiels, est régulier. Cependant, cet algorithme devient irrégulier quand les données sont variables, comme dans le cas du calcul formel. Supposons que l'on veuille calculer le produit scalaire de matrices formelles. L'algorithme proposé est alors hautement irrégulier, en fonction du volume de calcul lié à chaque matrice. Avec Athapascan-0a, nous utilisons le principe de décomposition procédurale parallèle pour décomposer un problème complexe en une collection de problèmes plus simples, qui peuvent être résolus en parallèle. Le régulateur de charge peut intervenir pour choisir une découpe, retarder des calculs et les placer au meilleur endroit.

Examinons d'abord comment le principe de décomposition procédurale parallèle peut être exprimé :

```
int par_scalprod(int n, vecteur v1, v2) {
  int r, r1, r2; /* résultats final et partiels */
  TaskId task1, task2; /* placements des sous-tâches parallèles */
  SynchroPt s1, s2; /* points de synchronisation intermédiaires */
  /* choix entre découpe parallèle ou calcul séquentiel */
  if (choice) {
      /* calcul parallèle: découpe */
      /* détermination des placements */
      task1 = ...
      task2 = ...
      s1 = Spawn(task1, SCALPROD#, "2D", n/2, v1, n/2, v2);
      s2 = Spawn(task2, SCALPROD#, "2D", n/2, v1+n/2, n/2, v2+n/2);
      /* attente des résultats partiels */
      WaitSpawnRes(s1, "d", &r1);
      WaitSpawnRes(s2, "d", &r2);
      /* combinaison en un résultat final */
      r = r1+r2;
  } else {
      /* calcul séquentiel */
      r = seq_scalprod(n, v1, v2);
  }
  return r;
}
```

FIG. 4.14 - *La décomposition procédurale parallèle.*

La procédure de calcul du produit scalaire suit la méthode de décomposition procédurale parallèle. Un choix est fait entre un calcul parallèle et un calcul séquentiel. Ce choix concerne le niveau de régulation de charge et ne sera pas abordé ici. Si la décision est en faveur d'un calcul parallèle (branche vraie), alors deux fonctions ScalProd sont appelées sur *task1* et *task2*, qui peuvent être sélectionnées par le niveau de régulation de charge, ou bien choisies au hasard, par exemple. Les points d'entrée distants sont spécifiés par leurs numéros (ici, *SCALPROD#*). Les paramètres sont envoyés avec chaque requête. Chaque opérateur *Spawn* retourne un point de synchronisation. Celui-ci est utilisé pour attendre le message de réponse de l'exécution distante. Le résultat partiel est alors reçu est accumulé. Si la décision n'est pas en faveur d'un calcul parallèle, le travail complet est effectué séquentiellement.

Le point d'entrée destiné à servir le calcul parallèle du produit scalaire peut être écrit comme suit :

```
EP_MODEL_MACRO (ScalProd, in, out)
BEGIN_EP
  int n; /* dimension des vecteurs */
  vecteur v1, v2; /* les vecteurs */

  /* récupération des paramètres */
  UnPack(in, "2D", &n, &v1, &n, &v2);
  /* spécification du résultat */
  Pack(out, "d", par_scalprod(n, v1, v2));
END_EP
```

FIG. 4.15 - *Le service ScalProd de calcul du produit scalaire.*

Le service ScalProd, introduit par le mot clé **EP_MODEL_MACRO**, a deux paramètres : le tampon de requête, *in*, et le tampon de réponse, *out*. Les paramètres du calcul sont déballés du tampon de

requête. Ensuite la méthode de décomposition procédurale parallèle est appliquée. Le résultat est emballé dans le tampon de réponse, *out*, et le processus léger exécutant le service est automatiquement terminé.

Le modèle de tâche supportant le point d'entrée SCALPROD pourrait être le suivant :

```
TASK_MODEL_MACRO (ScalProdTask, argc, argv) {
  int ntasks;
  TaskId *tasks;        /* nombre et identités des tâches participant au programme */
  /* les fonctions utiles */
  int seq_scalprod(...) {}
  int par_scalprod(...) {}
EP_MODEL_MACRO (ScalProd, in, out)
    /* voir ci-dessus */
END_EP
EP_MODEL_MACRO (InitTask, in, out)
    /* récupération des identités des autres tâches participant au programme */
    UnPack("T", ntasks, tasks);
    /* déclaration du point d'entrée */
    NewEntryPoint(SCALPROD#, ScalProd, ATL_UNBOUND, NULL);
END_EP
EP_MODEL_MACRO (TermTask, in, out)
    /* rien à exécuter ici */
END_EP
END_TASK
```

FIG. 4.16 - *Le modèle de tâche supportant le service ScalProd.*

Ce modèle de tâche définit le code du service de calcul, comme précédemment évoqué. Le service **InitTask** récupère les identités des tâches participant à la découpe procédurale. Ces identificateurs sont conservés dans la mémoire partagée de la tâche pour être accessibles à partir de la couche de régulation de charge. Le point d'entrée utilisateur est ensuite déclaré. Le service **TermTask** n'effectue aucune action particulière.

La section programme pourrait être la suivante :

```
PROGRAM_MODEL_MACRO (ScalProdProg, argc, argv) {
  int nprocs; /* nombre de processeurs virtuels */
  char **procs; /* les processeurs virtuels */
  TaskId *tasks; /* identités des tâches participant au programme */
  int n; /* dimension des vecteurs */
  vecteur v1, v2; /* les vecteurs */
  int r; /* résultat final */
  /* les fonctions utiles */
  int seq_scalprod(...) {}
  int par_scalprod(...) {}
  /* la tâche racine participe à la découpe parallèle;
  elle sert donc aussi le service ScalProd */
EP_MODEL_MACRO (ScalProd, in, out)
   /* voir ci-dessus */
END_EP
BEGIN_PROGRAM

  /* déclaration du point d'entrée */
  NewEntryPoint(SCALPROD#, ScalProd, ATL_UNBOUND, NULL);
  /* trouver les processeurs utilisables */
  GetProcs(&nprocs, &procs);
  /* démarrer une tâche par processeur (proc 0 = section programme) */
  tasks = GlobalMalloc(sizeof(TaskId)*nprocs);
  tasks[0] = GetTaskId();
  for (i=1; i<nprocs; i++)
    tasks[i] = LoadPVMTask(procs[i], "", "scalprod_task");
  /* initialiser les tâches en les informant des identités des autres */
  for (i=1; i<nprocs; i++)
    Call(task[i], SERV_INIT_TASK, "T", nprocs, tasks);
  /* lecture des vecteurs */
  ...
  /* appeler le calcul scalaire en local */
  r = par_scalprod(n, v1, v2);
  /* utilisation du résultat */
  ...
  /* terminer les tâches
  for (i=1; i<nprocs; i++)
    Call(task[i], SERV_TERM_TASK, "", "");
END_PROGRAM
```

FIG. 4.17 - *Le programme du produit scalaire.*

La tâche racine trouve les processeurs physiques de la machine virtuelle, y démarre des processeurs virtuels (tâches) et identifie chaque tâche vis à vis des autres. La tache racine participe ici à la découpe parallèle ; elle sert donc elle aussi un point d'entrée ScalProd. Le produit scalaire est appelé localement, mais se développera en parallèle, sous le contrôle d'un éventuel régulateur de charge. Lorsque l'exécution est finie, toutes les tâches filles sont terminées.

La découpe récursive est ainsi bien visible : le point d'entrée est appelé successivement sur différentes tâches, à chaque fois avec un problème plus petit, jusqu'à ce que le calcul puisse se faire séquentiellement. Si cette découpe possède un fort surcoût pour des données régulières, ce n'est plus vrai quand les composants des vecteurs sont par exemple des matrices formelles à cœfficients polynomiaux, auquel cas on ne peut pas faire de découpe statique garantissant un équilibrage de la charge. Notons que la transmission des paramètres peut être améliorée en transmettant des *pointeurs* sur les

données. Cela éviterait des transferts inutiles des données, en cascade. La version Athapascan-0a n'offre pas directement le concept de pointeur global, ni d'accès mémoire distant, mais ces concepts sont réalisables par le programmeur en définissant un service permettant l'accès aux données. D'autre part, l'emballage et l'appel des primitives Athapascan-0a est relativement coûteux si l'appel doit être local (ie. l'appelé s'exécute sur la même tâche que l'appelant). Il n'existe pas de solution miracle à ce problème, car l'emballage effectué par Athapascan-0 suit celui pratiqué par PVM. Le programmeur averti souhaitant optimiser son programme codera donc un harnais destiné à traiter le cas d'appel local spécialement, comme dans l'exemple :

```
if (task == GetTaskId()) {
  /* l'appel est local */
  r = par_scalprod(n, v1, v2);
} else {
  /* l'appel est distant */
  Call(task, SCALPROD#, "2D", n, v1, n, v2, "d", &r);
}
```

FIG. 4.18 - *L'appel local.*

Un harnais permet de traiter spécialement l'appel local. Dans ce cas, la transmission par valeur habituellement effectuée lors de l'emballage / déballage des paramètres peut souvent être transformée en une transmission par référence, évitant des copies de données.

Athapascan ne construit pas automatiquement des talons clients et serveurs à partir d'une description des services, comme le fait par exemple IDL. Cependant, il est possible d'écrire de tels talons manuellement, comme nous l'avons fait ici. Ces talons sont un peu différents des talons que l'on a l'habitude de manipuler, avec RPC-GEN par exemple, car ils intègrent la découpe parallèle. La fonction de base est une fonction séquentielle, comme *seq_scalprod* ici. Le talon client correspond au choix de découpe (exécution parallèle ou séquentielle). C'est la fonction *par_scalprod* du service de calcul. Le talon serveur correspond au service tel que nous l'avons écrit : déballage des paramètres, appel de la découpe procédurale parallèle, emballage du résultat.

Un point important à retenir de cet exemple est que l'algorithme est complètement indépendant de la politique de régulation de charge employée. La portabilité de l'application est donc garantie, sous réserve que la régulation soit efficace.

4.6 Réalisation de l'exécutif Athapascan-0a

Pour la réalisation d'Athapascan-0a, un premier choix concernait le noyau de multiprogrammation et le noyau de communication sous-jacents à utiliser. Nous allons détailler ici ces deux points, puis nous présenterons l'implantation qui a été réalisée au dessus de PVM et de différents noyaux de multiprogrammation.

4.6.1 Noyaux de multiprogrammation

Nous allons décrire le choix du noyau de multiprogrammation virtuel que nous avons fait, puis nous résumerons les demandes d'Athapascan-0a sur ce noyau et enfin nous indiquerons quels noyaux ont été effectivement utilisés dans Athapascan-0a.

4.6.1.1 Choix du noyau de multiprogrammation virtuel

A l'époque du début de cette thèse, il n'y avait pas de standard établi de noyau de multiprogrammation (le standard POSIX a été normalisé vers la fin de ce travail). En conséquence, nous avons préféré nous restreindre à un minimum de fonctionnalités présentes dans tous les noyaux de multiprogrammation et utiliser pour chaque architecture le noyau ad- hoc présent sur l'architecture. Il eut été possible de définir et porter un noyau unique ; cependant cela aurait nécessité un travail important dont la finalité en terme d'objectif de recherche était faible (de tels noyaux portables existant déjà sur certaines architectures). D'autre part, le fait de « dégrossir » les noyaux de multiprogrammation existants pour n'utiliser que la partie qui semblait essentielle était un objectif de recherche, par la définition de la partie essentielle justement, et la validation de cette définition.

4.6.1.2 Exigences d'Athapascan-0a sur le noyau de multiprogrammation

Dans le cadre d'Athapascan, le noyau de multiprogrammation doit permettre de créer un nombre quelconque, borné, de fils d'exécution. On doit pouvoir attacher un contexte à chaque fil d'exécution. Le noyau de multiprogrammation doit aussi offrir un moyen d'implanter des sémaphores : soit en offrant directement des sémaphores, soit par l'intermédiaire des primitives *suspend* et *resume*, soit par toute autre méthode (variables de condition + verrous, par exemple...). Comme nous l'avons dit dans la section sur les choix, nous utilisons des fils d'exécution en mode coopératif. Avec cette faible demande de fonctionnalités, n'importe quel noyau de multiprogrammation peut convenir.

4.6.1.3 Les noyaux de multiprogrammation employés

Chaque machine possède son propre noyau de multiprogrammation. Nous avons défini une couche d'interface standard avec les noyaux de multiprogrammation, offrant comme fonctionnalités : la création, le suicide et l'identification de processus léger, le maintien d'un contexte par fil d'exécution, et enfin la synchronisation entre fils d'exécution, par sémaphores. Nous avons utilisé des noyaux préemptibles ou non préemptibles. Dans le cas de noyaux préemptibles, tous les processus légers que nous créons sont au même niveau de priorité et gérés par un ordonnancement FIFO. Cela est suffisant pour inhiber la préemption. Tous les fils d'exécution utilisent des piles de même taille, sur les systèmes pour lesquels cela a une importance. Cette condition nous permet d'effectuer une petite optimisation décrite plus loin. Puisque les fonctionnalités demandées sur le noyau de multiprogrammation sont minimales, n'importe quel noyau peut être employé. Les noyaux utilisés sont recensés en 4.6.4.1 .

4.6.2 Noyaux de communication

Comme pour les noyaux de multiprogrammation, nous allons décrire le choix du noyau de communication que nous avons fait, quels sont les demandes qu'Athapascan-0a pose sur ce noyau et enfin nous présenterons le noyau utilisé, PVM.

4.6.2.1 Choix du noyau de communication

En ce qui concerne le noyau de communication, différentes approches étaient possibles. Il aurait été possible de bâtir ce travail directement au-dessus des fonctionnalités de communication d'un système d'exploitation, par exemple les *sockets* d'Unix [Stevens 1990, Comer & Stevens 1993]. Cette approche n'a pas été retenue car il est nécessaire de gérer des processeurs virtuels, ce qui nécessite un mécanisme portable, en supplément de la plupart des fonctionnalités de communication des systèmes

d'exploitation. De plus, ce genre de réalisation est un objectif de recherche en soi, déjà illustré par de nombreux exemples. Il aurait pu être possible de descendre au-dessous de l'interface du système d'exploitation pour définir un pilote spécifique, particulièrement pour exploiter un matériel spécifique. Cependant, nous n'avions aucun matériel spécifique au début de ce travail, et ce genre d'approche, très coûteuse en temps de développement qui plus est, allait à l'encontre de l'objectif de portabilité que nous nous étions fixé. La troisième approche consistait à utiliser l'un des systèmes de communication portables, gérant des processeurs virtuels, déjà développés par d'autres équipes de recherche. Parmi tous les choix possible, nous avons décidé d'opter pour PVM, puisque celui-ci correspondait à nos attentes et promettait à l'époque d'être un standard de fait pour les utilisateurs de machines parallèles. Cela nous permettait de prendre un système fonctionnel et de le modifier pour évaluer nos propres objectifs de recherche. De plus, PVM par sa portabilité, son efficacité relative, sa simplicité d'utilisation et son modèle de programmation à tampon infini, correspondait bien à ce que nous recherchions. PVM a été décrit en détail en 3.2.

4.6.2.2 Exigences d'Athapascan-0a sur le noyau de communication

L'exécutif d'Athapascan-0a se base sur le noyau de communication pour réaliser les tâches suivantes :

- administration de la Machine Virtuelle : toutes les machines réunies sous le contrôle d'Athapascan-0a forment une unique Machine Virtuelle. Athapascan-0a s'attend à pouvoir, dynamiquement, agrandir la Machine Virtuelle en s'étendant sur de nouvelles machines physiques, ou au contraire, expulser certaines machines physiques de la Machine Virtuelle.
- administration des Processeurs Virtuels : un Processeur Virtuel est attaché à une machine physique (sans possibilité de migration), c'est la responsabilité du noyau de communication de réaliser cet attachement ; le noyau de communication devant aussi fournir une identification globale unique pour chaque Processeur Virtuel.
- communications point à point : le noyau de communication doit être capable d'effectuer des envois de messages fiables, avec une cohérence processeur (voir en 2.2.3.2), sans limite de taille (jusqu'à celle de la mémoire du site), entre deux Processeurs Virtuels. L'émission doit être de préférence non bloquante, et la réception bloquante ou non bloquante, non sélective, avec tamponnage des messages en attente de réception. De plus, les primitives de communication doivent accepter de travailler avec un nombre indéfini de messages en cours de formation, à envoyer, et en cours de lecture, à recevoir.
- gestion de l'hétérogénéité : Athapascan-0a gère une file de valeurs typées dans chaque message. La Machine Virtuelle pouvant être hétérogène, le noyau de communication doit fournir une « machine à file » pour l'emballage et le déballage des types de base : *int*, *long*, *double*, *char* et *byte*.

Actuellement, PVM 3, ou tout noyau respectant l'interface de PVM 3, peut être employé comme noyau de communication. Il est possible que, contrairement à PVM, de futurs noyaux de communication n'offrent pas de fonctionnalités d'administration de la machine virtuelle, ou des processeurs virtuels. Auquel cas ces fonctionnalités seraient reprises en charge par l'exécutif Athapascan-0.

4.6.3 Implantation d'Athapascan-0a

Enumérons les différentes implantations possibles avec PVM :

- soit construire l'application Athapascan-0a dans les démons de PVM. Un fil d'exécution aurait réalisé la fonction du démon (brassage des messages) et d'autres fils auraient effectué les

services. Il aurait été question en quelque sorte de rajouter des points d'entrée au démon. Cependant cela aurait nécessité :

- de modifier le démon et donc de casser la portabilité
- d'utiliser des fils d'exécution préemptibles de façon à ce que le démon de communication tourne assez souvent. Là encore, cela aurait réduit la portabilité et augmenté la difficulté d'implantation.

- soit modifier la bibliothèque de PVM pour la rendre synchrone vis à vis des fils d'exécution et utiliser des fils d'exécution en temps partagé. Cette approche est celle retenue par PM^2 (décrit en 5.2). Les défauts sont du même ordre que précédemment. D'autre part, le fonctionnement en temps partagé n'était pas notre souhait majeur.
- soit adjoindre à la bibliothèque de PVM un noyau de fils d'exécution coopératifs. Cela peut être fait sans modification de PVM comme nous le verrons. C'est cette approche que nous avons retenue, car elle est simple à mettre en œuvre et reste très facilement portable.

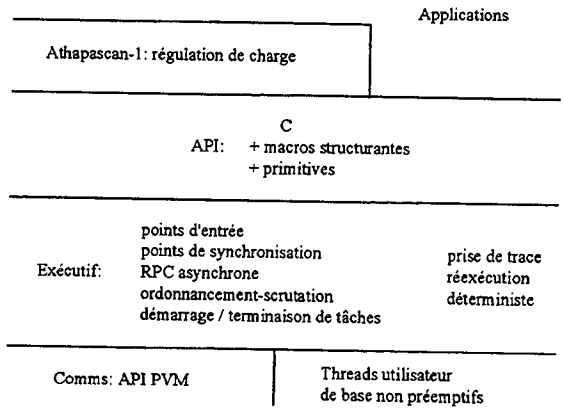

FIG. 4.19 - *L'organisation d'Athapascan-0a.*

Athapascan-0a est basé sur un noyau de communication qui respecte l'API de PVM et sur un noyau de multiprogrammation « virtuel ». Les deux couches d'Athapascan-0a sont montrées, l'exécutif et l'interface de programmation. Les applications peuvent employer le niveau Athapascan-1, une couche supplémentaire de régulation de charge, ou directement Athapascan-0.

Dans la suite nous allons décrire en détail l'implantation d'Athapascan-0a. Nous commencerons par une description de l'interface entre Athapascan-0a et les différents noyaux de multiprogrammation légère. Puis nous décrirons l'interface entre Athapascan-0a et un noyau de communication qui implante l'API de PVM. Enfin nous montrerons comment les notions de points d'entrée et d'appel RPC asynchrone sont implantées dans Athapascan-0a, comment est réalisé le cœur d'Athapascan-0a - la progression des calculs et des communications - c'est à dire l'opération d'ordonnancement-scrutation et enfin comment l'initialisation et la terminaison d'un programme Athapascan-0a sont effectuées. Nous terminerons par une récapitulation des différents portages réalisés.

4.6.3.1 Interface entre Athapascan-0a et la multiprogrammation légère

Nous allons décrire les fonctionnalités retenues pour la couche d'interface avec les noyaux de multiprogrammation ; nous indiquerons aussi celles parmi les plus communes qui nous ont semblé trop difficiles à supporter ou bien sans intérêt pour Athapascan-0a. Nous détaillons ensuite les informations maintenues par Athapascan-0a pour chaque fil d'exécution. Nous présentons aussi une

optimisation de réutilisation rendue possible par l'utilisation d'une unique taille de pile. En dernier lieu, nous décrivons quelques considérations pour l'implantation.

Les fonctionnalités Comme nous l'avons dit précédemment, Athapascan-0a utilise différents noyaux de processus légers, tous dans un mode coopératif. Cette assertion simplifie grandement la tâche de réalisation. En particulier, il n'y a pas de problème avec l'utilisation d'une bibliothèque de fonctions quelconque, puisque de toute façon l'exécution est effectuée en exclusion. Ce ne serait pas le cas si les fils d'exécution étaient préemptibles. Il aurait alors fallu utiliser des bibliothèques réentrantes et sauves vis à vis des processus légers.

Les fonctionnalités de la couche d'interface avec les noyaux de processus légers sont les suivantes :

- initialiser le noyau de fils d'exécution (spécification de la taille de pile à utiliser),
- sortir du programme (code d'erreur à retourner),
- créer un fil d'exécution (adresse de la fonction de service à exécuter),
- suicide d'un fil d'exécution,
- identifier le fil d'exécution courant et surtout maintenir un contexte privé au fil d'exécution,
- gérer des sémaphores : création, opérations P et V,
- informer du nombre de fils d'exécution actifs (non bloqués), et du nombre total de fils d'exécution.

Notons que certaines fonctionnalités courantes sont absentes, par exemple :

- le *join* (l'attente du résultat d'un fils),
- le *yield* (le fait de donner la main à un autre fil d'exécution prêt sans se bloquer sur une condition d'attente),
- le *cancel* (le meurtre d'un fil d'exécution) et le rappel de nettoyage qui peut s'en suivre,
- les priorités d'exécution des fils,
- la gestion des signaux... le comportement du noyau sous-jacent est répercuté.

Parmi ces fonctionnalités inutilisées, celles qui pourraient être ajoutées sont les priorités et le *yield*, ou peut être mieux le *yield* dirigé, c'est à dire la commutation vers un fil d'exécution prêt *désigné*. Ces deux fonctionnalités pourraient être utiles pour contrôler plus précisément la gestion de l'ordonnancement des fils d'exécution. Les fonctionnalités de *join* et de *cancel* sont inutiles dans la réalisation que nous avons faite. La gestion des signaux dépend fortement de la machine cible et est très secondaire dans le modèle de calcul d'Athapascan-0.

Les informations maintenues pour chaque fil d'exécution Athapascan-0a a besoin de maintenir une certaine quantité d'informations pour chaque fil d'exécution. Ces informations sont stockées dans une zone globale. Cette zone contient pour chaque fil d'exécution :

- l'identité du fil d'exécution pour le noyau de processus légers sous-jacent,
- les identités des tampons de communication, en réception et envoi,
- des informations pour une optimisation d'exécution :
  - un indicateur d'initialisation,
  - l'adresse de la fonction de service a exécuter,
  - un « tampon de saut » (*jumpbuf*) pour les changements de contexte,
  - un sémaphore pour bloquer le fil d'exécution,
- des informations pour le déverminage, la prise de traces et la réexécution déterministe :
  - le numéro du point d'entrée servi par ce fil d'exécution,
  - le surnom donné,

- l'identification de la tâche et du fil d'exécution appelant,
- la position dans la séquence pour la réexécution déterministe,
- enfin des informations de mesure de temps.

Cette zone a une dimension fixée. Elle est actuellement représentée de façon contiguë, sous forme d'un tableau. L'indice dans ce tableau permet d'identifier un fil d'exécution de façon unique pour Athapascan-0a. Les parties libres sont gérées en une liste chaînée. Il est ainsi immédiat de trouver un emplacement pour un nouveau fil d'exécution. La zone pourrait être dynamiquement réallouée avec une dimension plus grande si la place venait à manquer. L'identification des fils d'exécution resterait valable après agrandissement. Actuellement cependant, la zone est de taille fixée assez grande (entre 100 et 1000, selon le noyau utilisé) et en cas d'épuisement, Athapascan-0a retourne une erreur fatale.

Lors de la création d'un fil d'exécution, il faut trouver une structure vide dans la zone, créer effectivement le fil d'exécution puis enregistrer certaines informations dans la structure. En particulier, la structure contient l'identité du fil d'exécution, telle que retournée par le noyau de processus légers. Cela permet d'agir sur un fil d'exécution, par l'intermédiaire du noyau de processus légers, en connaissant son identification pour Athapascan-0a. A l'inverse, lors d'un changement de contexte, il est nécessaire de retrouver les informations associées au fil d'exécution nouvellement (ré-)activé. Pour cela, l'identité du fil d'exécution pour Athapascan, son index dans le tableau précité, est enregistré dans le contexte privé du fil d'exécution, géré par le noyau de processus légers. Il y a donc bijection entre l'identité au niveau du noyau de processus légers et l'identité considérée par Athapascan. Il aurait été possible de stocker toutes les informations de chaque fil d'exécution dans son contexte privé. Cela aurait permis de considérer pour Athapascan l'identification retournée par le noyau de processus légers (en général c'est un pointeur). Cependant, soit la structure associée au contexte privée était allouée dynamiquement (donc avec un coût important à la création), soit elle était puisée depuis un tableau de structures, ce qui revient au même que la solution choisie. Pour le déverminage, il était plus simple de considérer l'index dans le tableau plutôt que le pointeur, c'est pourquoi la solution décrite a été choisie.

L'optimisation de réutilisation Dans la philosophie d'Athapascan, l'utilisation normale des processus légers est de créer un fil d'exécution, qui va exécuter une fonction de service puis se suicider. Dans Athapascan-0a, une restriction est posée : tous les fils d'exécution possèdent des piles de même taille. Cette considération nous permet d'effectuer une petite optimisation, consistant à ne pas suicider le fil d'exécution se terminant, mais à le mettre en attente pour une prochaine fonction de service à exécuter. On réduit ainsi le coût de création d'un fil d'exécution à un simple coût de changement de contexte. Notons que pour la plupart des noyaux de processus légers, le contexte « privé » d'un fil d'exécution ne peut être modifié que par lui-même. Pour rattacher une information à un fil d'exécution, il est donc nécessaire que celui-ci exécute d'abord un harnais, qui modifie son contexte privé, puis lance la fonction de service à effectuer. La méthode du harnais est donc systématiquement employée. Ce harnais est mis à contribution pour l'optimisation de création : l'algorithme consiste à éternellement établir un point de saut, exécuter la fonction de service demandée si l'on ne revient pas du point de saut, puis se bloquer sur un sémaphore avant de recommencer. Lorsque le fil d'exécution se suicide, il chaîne sa structure d'information dans la file des fils libres, puis saute sur le point de saut établi à la création. Ce saut non-local le ramène sur le blocage de son sémaphore. Lorsqu'un nouveau fils d'exécution doit être créé, une structure libre est choisie et soit l'indicateur d'initialisation nous informe que le fil d'exécution n'a jamais été créé, auquel cas on en crée un qui exécute le harnais d'abord, soit le fil d'exécution est déjà créé et bloqué en attente sur son sémaphore, auquel cas il suffit de le réveiller. L'algorithme correspondant est détaillé plus formellement ci-dessous ; nous donnons d'abord l'algorithme sans optimisation pour illustrer l'utilisation du harnais, puis la version optimisée pour la réutilisation.

```
création d'un fil d'exécution (adr_fonction) :
enlever un emplacement de la liste des libres,
y enregistrer l'adresse de la fonction que doit exécuter le fil,
créer un fil d'exécution, lui dire d'exécuter le harnais avec comme paramètre son index.
harnais (index) :
stocker l'index dans le contexte privé du fil d'exécution
exécuter la fonction identifiée grâce à l'index
se suicider.
suicide :
mettre l'emplacement dans la liste des libres,
suicider le fil d'exécution.
```

FIG. 4.20 - *Algorithme de création / suicide sans optimisation.*

Cet algorithme illustre l'utilisation du harnais.

```
création d'un fil d'exécution (adr_fonction) :
enlever un emplacement de la liste des libres,
y enregistrer l'adresse de la fonction que doit exécuter le fil,
si l'emplacement n'est pas initialisé :
  l'emplacement est maintenant initialisé,
  créer un fil d'exécution, lui dire d'exécuter le harnais avec comme paramètre son index,
sinon :
  signaler le sémaphore associé au fil d'exécution.
harnais (index) :
stocker l'index dans le contexte privé du fil d'exécution
pour toujours :
  établir un point de saut ici
  si on ne revient pas du point de saut, exécuter la fonction
  se bloquer sur le sémaphore associé au fil d'exécution.
suicide :
mettre l'emplacement dans la liste des libres,
sauter au point de saut pour se bloquer en attente de travail.
```

FIG. 4.21 - *Algorithme de création / suicide avec optimisation de réutilisation.*

Le harnais est maintenant employé pour réutiliser les fils d'exécution, sous la contrainte que la taille de pile soit constante. Le saut non local est réalisé à l'aide des primitives setjmp et longjmp d'Unix.

4.6.3.2 Interface entre Athapascan-0a et PVM

L'interface entre Athapascan-0a et un noyau de communication respectant l'interface de PVM est caractérisée par la gestion des machines physiques et virtuelles, la gestion de l'hétérogénéité de format des données, et surtout l'utilisation des communications point à point. Nous décrivons chacun des ces aspects. La notion de groupe de PVM n'est absolument pas utilisée, de même que la signalisation entre tâches.

Machines physique et virtuelle Athapascan-0a repose fortement sur PVM en ce qui concerne l'administration de la machine physique et des processeurs virtuels. En effet, Athapascan-0a propose une interface directement dérivée de celle de PVM pour la gestion de processeurs physiques (ajout, suppression de processeurs, identification des processeurs).

Pour les processeurs virtuels, les tâches, Athapascan-0a fait correspondre sa notion de tâche avec celle de PVM, jusqu'à utiliser l'identification de PVM pour sa propre identification des tâches. La création d'une nouvelle tâche se fait d'une façon analogue à ce qui se fait dans PVM. La création d'une tâche n'a jamais été considérée comme une opération cruciale et est donc quelque peu imparfaite. On ne peut créer qu'une tâche fille à la fois, de façon synchrone. D'autre part, il n'est pas possible de raffiner les arguments de cette tâche. Les arguments transmis selon le mécanisme Unix traditionnel de lancement de processus sont identiques à ceux de la tâche racine (avec un complément établi automatiquement par l'exécutif, destiné au démarrage de la tâche fille). Ce comportement permet en particulier d'employer de façon transparente des options d'exécution (par exemple *toutes* les tâches peuvent s'exécuter dans un mode particulier). Il est cependant possible de raffiner le comportement de la tâche fille lors de son initialisation (à l'aide du point d'entrée InitTask). Les tâches filles sont insérées dans une liste gérée par la tâche parente. Chaque tâche fille connaît aussi l'identification de sa tâche parente. Cette structure doublement chaînée distribuée permet de terminer proprement un programme : lors de la production d'une erreur, un message de terminaison est envoyé sur toutes les tâches filles et la tâche parente. Ce message est répercuté par les tâches qui le reçoivent, sur leur filles et leur parente (sauf s'il en provient). En conséquence toutes les tâches sont terminées. Lors de cette remontée vers la tâche racine, un message d'erreur en clair peut être transmis, permettant une meilleure information de l'utilisateur (nature et tâche source de l'erreur).

Hétérogénéité Une autre fonctionnalité importante de PVM reprise par Athapascan-0a est la gestion de l'hétérogénéité. Athapascan-0a peut simplement être compilé en mode homogène (pas de conversion de format) ou hétérogène (utilisation de XDR). Athapascan-0a ne fait que répercuter l'information sur PVM, qui fait tout le travail. Notons que cette organisation est logique, les concepteurs de PVM proposant à terme une décision adaptative de la conversion de format [Zhou & Geist 1995*b*], qui échapperait donc de toute façon à Athapascan-0a...

Communications point à point La dernière fonctionnalité de PVM qu'Athapascan-0a utilise intensivement est la communication point à point. Athapascan-0a utilise le modèle de communication de PVM, l'emballage dans des tampons de communication, avec un encapsulage adapté au modèle de programmation par RPC.

A ce fonctionnement très proche de PVM ont été ajoutées quelques extensions :

- il est possible de spécifier que toutes les communications de données seront « sécurisées » par une information de type précédant la donnée. De cette façon, il est possible de détecter toute incohérence de type entre l'émission et la réception. Sans aller aussi loin qu'un typage fort et général comme ceux que l'on trouve dans les langages orientés-objets (qui nécessite la participation du compilateur), ce test de conformité permet de déterminer beaucoup d'incohérences (par exemple inversion de l'ordre de deux données de types différents dans un message). Seul le type de base est pris en compte ; par exemple si l'on envoie un tableau de n entiers, il peut être légalement reçu comme un unique tableau de n entiers, comme plusieurs tableaux d'entiers (totalisant n éléments), ou encore comme n entiers distincts, ou un mélanges de tableaux et d'entiers... Ce test est une option d'exécution, habituellement désactivée lors de l'utilisation normale.
- contrairement à PVM, il est aussi possible de spécifier des données à transmettre qui ne sont pas référençables (constantes, résultats de calcul). Cela tend à simplifier les programmes, puisqu'autrement, il faut définir des variables pour stocker chaque donnée à transmettre.
- comme dans les dernières versions de PVM, il existe des primitives de « plus haut niveau » permettant de simplifier l'écriture des programmes par des formats à la *printf*. L'exécutif

Athapascan-0a peut allouer lui-même les tableaux reçus, simplifiant de ce fait la programmation : il est inutile de spécifier la réception de la taille, l'allocation du tableau, la réception des données du tableau ; ces trois opérations peuvent être faites par l'exécutif, avec la spécification d'un seul « format ».

Athapascan-0a autorise le mode *direct-route* de PVM version domaine public comme option d'exécution. Par défaut les démons PVM sont sur le chemin de communication (mode *direct-route* désactivé). Il est aussi possible d'utiliser le mode *data-inplace* de PVM comme option d'exécution. L'exécutif supporte pleinement ce mode, mais l'utilisateur n'a pas le droit d'employer dans ce mode des communications sur des données non référençables. Dans les deux cas, l'utilisation est un peu stricte puisque ce sont des options globales : on ne peut pas spécifier ces options pour une seule communication. En revanche, le gain (en particulier de *direct-route* sur IP) est pleinement répercuté sur les applications écrites en Athapascan-0a.

4.6.3.3 Implantation des services

L'implantation d'un service est réalisée en deux parties. Une tâche cliente effectue l'appel d'un service offert par une tâche serveur ; la tâche serveur définit et sert le service proposé. Nous allons d'abord décrire comment le contrôle de flux peut être effectué ; ensuite quelles sont les différentes étapes d'un appel de service, le format des messages échangés et les structures de données utilisées ; nous terminons par la définition et l'exécution d'un service.

Contrôle de flux En prélude à la description de l'implantation des services au dessus de PVM, nous devons insister sur le fait que PVM ne permet pas la réception de messages par interruption. La seule fonctionnalité possible est donc de *scruter* à certains moments le réseau. Cette scrutation prend un certain temps ; il ne faut donc pas la faire trop souvent. Dans le modèle de communication de PVM, les messages reçus entre deux scrutations sont tout simplement tamponnés par le démon de la tâche réceptrice, ceci de façon transparente. Notons que PVM domaine public n'effectue pas de contrôle de flux. Le tamponnage s'effectue donc jusqu'à l'explosion de la mémoire (de l'espace de pagination sur disque). D'où un comportement pour l'application assez indéterministe et difficile à prévoir. En conséquence les utilisateurs doivent concevoir leur propre contrôle de flux. Dans un programme monoprogrammé, c'est relativement simple si les correspondants sont synchronisés par des échanges du type envoi- réception. Un correspondant ne peut pas émettre plus d'un message sans se bloquer en attente de son partenaire. Dans un programme client-serveur multiprogrammé, les clients peuvent générer un grand nombre de requêtes ; deux types de problèmes peuvent provenir de ce comportement :

- les messages ne sont pas reçus assez vite, en conséquence la capacité de tamponnage est épuisée - l'espace de pagination déborde - et le processus démon du serveur est tué par le système Unix qui n'apprécie pas cette situation ;
- les requêtes sont reçues suffisamment vite, mais consomment la totalité des ressources du serveur (fils d'exécution disponibles par exemple). Le serveur doit alors refuser certaines requêtes. Le client peut, après un délai destiné à permettre le désengorgement du serveur, réitérer sa requête. Ce protocole peut être implicite dans la réalisation du RPC, c'est à dire pris en charge par l'exécutif, ou bien explicite, c'est à dire pris en charge par le programmeur.

Nous ignorons ici les possibilités de perte de messages, c'est à dire que nous supposons le noyau de communication fiable. Nous ignorons de même le cas limite où l'émission d'un message de refus est rendue impossible par l'engorgement du serveur. Le premier problème (épuisement de l'espace tampon) a peu de solutions pratiques : à moins de définir une périodicité de scrutation très courte,

il est difficile d'assurer qu'un comportement chaotique de l'application ne va pas faire exploser la mémoire. De plus, qui dit périodicité dit activation sur un signal de fin de délai, or les signaux sont des mécanismes coûteux dans Unix. La seule solution possible en l'état (c'est à dire sans changer le noyau de communication pour intégrer un contrôle de flux ou bien un mécanisme de réception par interruption) se révèle donc inadaptée et inefficace. Le choix que nous avons fait est de laisser la résolution de ce problème à l'application : c'est au programmeur de réaliser un contrôle de flux tel qu'il n'y ait pas trop de messages générés. Ce choix nous semble consistant avec l'optique d'utilisation d'Athapascan-0 dans le cadre du projet APACHE : la couche Athapascan-1 doit de toute façon adapter le grain de découpe à la machine cible. Il nous semble naturel que cette adaptation conduise à un contrôle de flux tel qu'une scrutation non périodique, réalisée par exemple à chaque blocage de fil d'exécution, soit suffisante. Le second problème (épuisement des ressources du serveur) est considéré comme identique au premier à partir du moment où les ressources en question sont conséquentes (par exemple, le nombre de fils d'exécution utilisables est supérieur à cent dans Athapascan-0a). La solution envisagée est donc la même : reporter le problème sur la couche Athapascan-1, avec une résolution simplifiée par le fait qu'une borne minimum peut être statiquement donnée pour toutes les ressources. Le niveau Athapascan-1 ayant le contrôle de la découpe et de l'ordonnancement, il lui est possible de ne pas épuiser les ressources disponibles. Dans la réalisation actuelle d'Athapascan-0a, l'épuisement des ressources d'un serveur correspond à une erreur qui ne peut être récupérée, mais qui est rapportée. Il n'y a donc même pas de refus de la part des serveurs.

Un refus pourrait cependant facilement être ajouté ; nous n'en avons pas senti la nécessité dans la mesure où les utilisateurs ne se sont pas plaints de cette absence ; si ce problème apparaît sur les tests jouets proposées, il est bien pris en compte comme une conséquence de la granularité dans les applications réelles. Notons à l'inverse qu'un refus de servir pourrait être à la base d'un mécanisme de régulation de charge (par exemple : décider au hasard d'une machine, jusqu'à ce que la requête soit acceptée. La charge serait alors distribuée en fonction des ressources disponibles). La couche Athapascan-1 n'ayant pas atteint ce niveau de développement, nous n'avons pas, encore une fois, ressenti la nécessité d'implanter un mécanisme de refus. Enfin, et dernière considération, le refus pourrait être stimulé par un manque de ressources initial (il n'y a plus de fil pour exécuter une requête) ou bien par un manque en cours d'exécution (il n'y a plus de mémoire disponible maintenant, au milieu de l'exécution d'un service). Refuser de façon similaire ces deux cas nous amène au problème bien connu de la réexécution d'un service partiellement exécuté. Les deux cas sont fondamentalement différents. Dans le premier, le service n'a pas encore démarré ; il est donc facile pour l'exécutif de le refuser pour tenter de le redémarrer plus tard. Le second cas est une erreur d'exécution, qui doit être traitée telle quelle par le programmeur ; c'est à lui de retourner une erreur compréhensible en guise de résultat d'exécution de la requête, et de tester cette réponse particulière. Ce n'est donc pas du ressort, à notre avis, de l'exécutif.

Appel d'un service Nous détaillons pas à pas la réalisation des différentes étapes d'un appel de service.

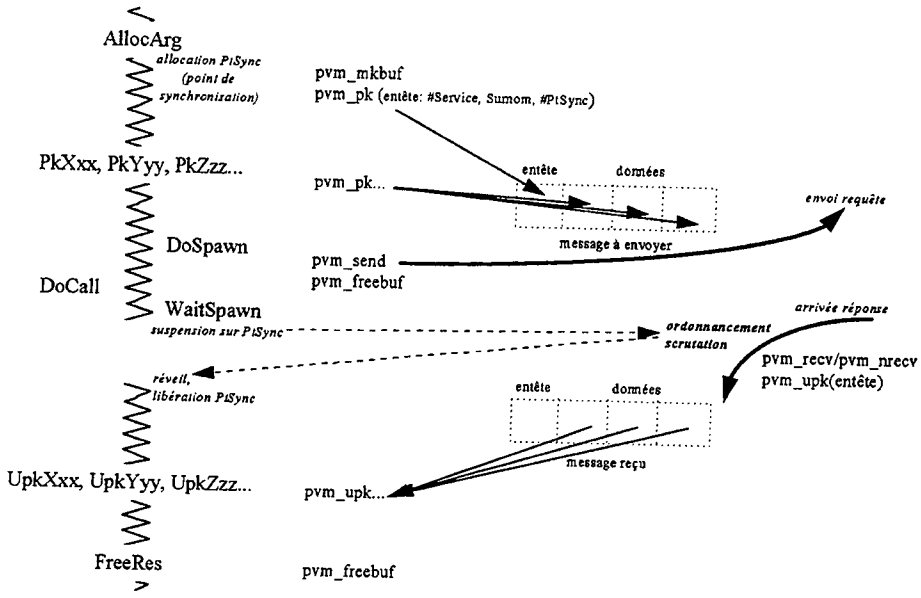

FIG. 4.22 - *Les étapes d'un appel d'un service.*

La primitive **AllocArg** consiste simplement à allouer un nouveau point de synchronisation et un nouveau tampon de communication et, PVM ne permettant pas de modifier ultérieurement les données stockées dans un tampon, à y enregistrer l'entête de la requête (le numéro du point d'entrée appelé, le *surnom*, l'identité du point de synchronisation utilisé pour recevoir la réponse). Les différentes primitives **PkXxx** permettent d'emballer des informations dans le tampon de communication, de façon très semblable à PVM. La primitive **DoCall** envoie le message ainsi construit, puis bloque le fil d'exécution sur le point de synchronisation. Alternativement, la primitive **DoSpawn** envoie simplement le message et fait apparaître à l'utilisateur le point de synchronisation intermédiaire. Dans les deux cas, le tampon de communication de PVM est libéré. La primitive **WaitSpawn** effectue l'attente bloquante sur le résultat. A la fin du **DoCall** ou du **WaitSpawn**, une opération d'ordonnancement-scrutation est exécutée pour recouvrir l'attente du message de réponse par du calcul et prendre en compte son arrivée. Une fois la primitive **DoCall** ou **WaitSpawn** terminée, l'exécutif a libéré le point de synchronisation et a associé le message reçu par PVM à un nouveau tampon de communication. A partir de celui-ci sont déballés les résultats par les différentes primitives **UpkXxx**. La primitive **FreeRes** permet de libérer, après utilisation, le tampon de résultats.

Format des messages Chaque message de requête est précédé d'un entête qui spécifie le numéro de point d'entrée appelé, le surnom associé et l'identité du point de synchronisation qu'il faudra retourner avec les résultats. Ces informations sont stockées sur deux entiers (le surnom et le point de synchronisation sont stockés chacun sur 16 bits). En mode de réexécution, l'entête est plus important puisqu'il contient aussi l'identité du service appelant, celle du fil d'exécution appelant et un numéro de séquence pour distinguer l'appel parmi les différents appels effectués par ce fil. Dans tous les cas, l'identité de la tâche appelante est fournie par PVM, qui l'extrait de son propre entête. Les messages de réponse ont la même structure : le numéro de point d'entrée est par convention 0 et il n'y a pas de surnom, seulement l'identité du point de synchronisation auquel est destiné la réponse. A la suite de l'entête, le message contient toutes les données paramètres ou résultats.

Les descripteurs de message Un descripteur de message décrit à l'intérieur de l'exécutif les messages de requête et de réponse qui sont manipulés. Il contient :

- la version du format dans lequel le message est rédigé,

- l'identité de la tâche où envoyer le message,
- l'identité du tampon PVM contenant le message,
- l'identité du point de synchronisation pour attendre une réponse à ce message,
- l'identité de la tâche à qui répondre,
- la référence à donner en réponse au message,
- des informations pour la réexécution déterministe,
- des informations d'instrumentation.

Les descripteurs de messages sont gérés à l'aide d'une liste de structures libres. Les descripteurs de messages sont alloués par blocs. Cela permet de ne pas en allouer un trop grand nombre d'un coup, tout en autorisant un grand nombre de descripteurs. De plus l'allocation est ainsi optimisée par rapport à une allocation séparée de chaque descripteur. Un descripteur de message est utilisé à la fois pour les messages en cours de construction, à envoyer, et pour les messages reçus. Certains champs ne sont utilisés que dans l'un des deux cas. Lors de la création d'un descripteur de message, pour envoi, un tampon de communication PVM est créé, et éventuellement, un point de synchronisation lui est alloué, si l'on prévoit d'attendre une réponse pour ce message.

Les points de synchronisation Un point de synchronisation permet l'attente d'un message de réponse. Il contient :

- un indicateur d'initialisation,
- un sémaphore pour se bloquer en attente,
- l'identité du fil d'exécution bloqué en attente,
- l'identité du message reçu en réponse,
- des liens permettant de construire des listes.

Les points de synchronisation sont aussi gérés à l'aide d'une liste de structures libres. Tous les points de synchronisation sont stockés dans une table, l'identité d'un point de synchronisation est son index dans la table. La table est actuellement de taille fixe, mais pourrait être réallouée dynamiquement, pour autoriser un nombre quelconque de point de synchronisation. L'indicateur d'initialisation sert à indiquer si le sémaphore référencé dans le point de synchronisation est valide ou non : le sémaphore n'est effectivement créé que quand le point de synchronisation est sur le point d'être utilisé pour la première fois. Les sémaphores sont donc réutilisés.

La définition des services Du point de vue du serveur, il s'agit dans un premier temps de définir le service ; c'est à dire principalement d'enregistrer le point d'entrée dans une table, puis dans un second temps de le servir, c'est à dire de récupérer des messages de requête pour ce service, d'instancier des fils d'exécution exécutant la fonction de service associée et enfin de récupérer le contrôle de l'exécution lorsque le fil se bloque, pour en activer un autre. Le cycle d'exécution est donc le suivant :

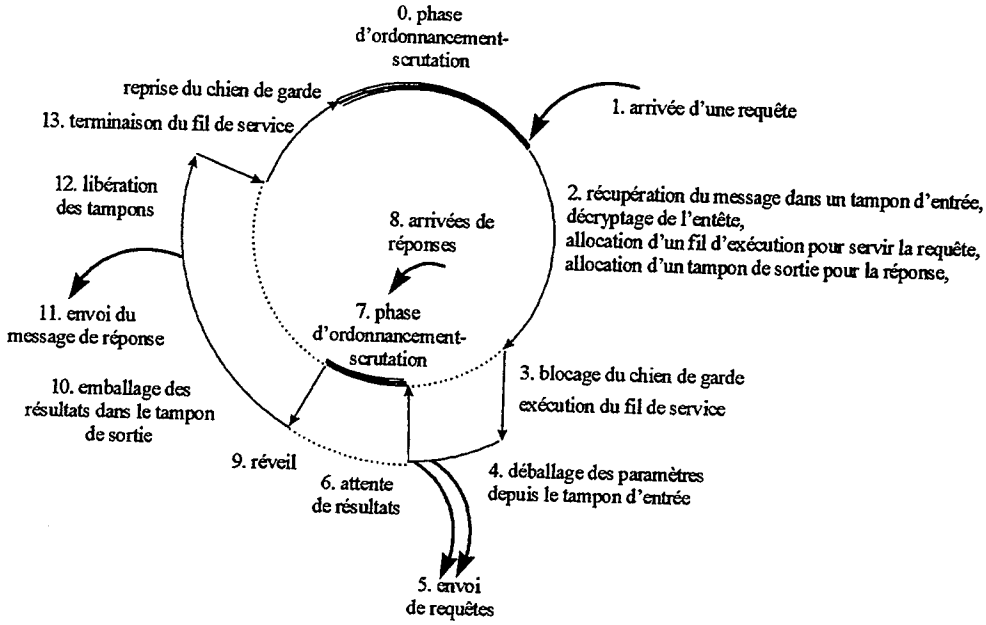

FIG. 4.23 - *Le cycle d'exécution d'un service.*

L'arrivée d'une requête conduit à la création d'un nouveau fil d'exécution pour un service donné. Celui-ci déballe ses paramètres du tampon d'entrée et emballe ses résultats dans le tampon de sortie. Eventuellement, il envoie des messages de requête, se bloque en attente de leurs résultats, et rend de ce fait la main à l'ordonnanceur. Lorsque le service est terminé, le fil d'exécution est détruit et le message de résultat envoyé. La phase d'ordonnancement- scrutation est effectuée à chaque fois qu'un fil exécutant un service se bloque ou se termine.

Ce cycle d'exécution correspond conceptuellement à deux fils d'exécution tournant à tour de rôle. L'un boucle sur un algorithme d'ordonnancement-scrutation, l'autre effectue un service. Le blocage de l'un permet à l'autre de s'exécuter. Le fil qui effectue l'ordonnancement- scrutation peut utiliser différents algorithmes ; par exemple il peut scruter systématiquement le réseau ou bien seulement quand aucun fil d'exécution n'est prêt. Ce fil sera appelé le *chien de garde* puisque c'est lui qui fait progresser les séquences de calculs. Notons que cette approche conceptuelle d'un chien de garde et de plusieurs fils de calcul a été utilisée telle quel dans les premières implantations d'Athapascan-0a ; cependant dans les dernières versions, les fonctionnalités du chien de garde, l'ordonnancement-scrutation, sont directement effectuées par le fil de calcul ; cela accélère un peu les traitements comme nous le verrons par la suite.

L'exécutif Athapascan-0a gère, sur chaque tâche, une table des services déclarés sur la tâche. Comme les numéros de point d'entrée sont pris dans un espace très large (32 bits), cette table est accédée par un adressage dispersé. La table contient, pour chaque service enregistré, un descripteur de service. Les descripteurs de service sont alloués par blocs, dynamiquement. Un descripteur de service contient :

- le numéro du service,
- l'adresse de la fonction de service associée,
- la limite de concurrence du service,
- la concurrence actuelle,
- une file pour les requêtes en attente (vide si la concurrence actuelle est inférieure à la limite de concurrence),

- le pointeur sur l'espace mémoire partagé par tous les fils exécutant ce service,
- le nom en clair du service (pour les traces),
- et certaines informations pour la réexécution déterministe.

Un nouveau service est déclaré par **NewEntryPoint**. Un service peut être supprimé (fermé). La gestion de la cohérence est cependant à la charge du programmeur. Quand l'exécutif ne trouve pas le service demandé par une requête, il provoque une erreur fatale.

Lorsque la limite de concurrence d'un service est dépassée, toutes les requêtes ultérieures sont stockées dans une file (par service). Lorsqu'un fil d'exécution termine l'exécution d'un service, il teste cette file et éventuellement en sort une nouvelle requête à exécuter. Ce fonctionnement permet de conserver le fil d'exécution en cours et ne demande pas de scrutation du réseau. D'autre part, les requêtes seulement sont mémorisées contrairement à ce qui se passerait si l'on démarrait toujours un fil d'exécution dès qu'une requête est reçue, qui se bloquerait alors sur un sémaphore pour respecter la limite de concurrence. Dans ce mode, il y aurait en mémoire, en plus des requêtes, les piles des fils d'exécution en attente. D'autre part, il y aurait aussi forcément un changement de contexte entre le fil d'exécution faisant redescendre la concurrence en dessous de la limite et celui exécutant la nouvelle requête. Enfin, il y aurait aussi forcément création d'un nouveau fil d'exécution. Notons que le fonctionnement choisi repose fortement sur le fait que la taille de la pile ne change pas entre les différentes requêtes.

4.6.3.4 Ordonnancement et scrutation

L'opération d'ordonnancement-scrutation a pour but de sélectionner le prochain fil d'exécution qui tournera. Cette sélection peut se faire parmi les fils d'exécution prêts à tourner, ou bien parmi les requêtes et les réponses que l'on peut lire sur le réseau. Il faut appliquer un certain algorithme d'ordonnancement-scrutation. Cet algorithme doit fournir un fil d'exécution prêt à tourner, ou bien indiquer la terminaison de la tâche en signalant le chien de garde. Différents algorithmes d'ordonnancement-
scrutation peuvent être employés, mais la base est la même : soit l'on sélectionne un fil d'exécution préexistant, soit l'on décide de scruter le réseau. Cette scrutation peut être non-bloquante si l'on veut pouvoir effectuer un autre ordonnancement dans le cas où le réseau n'offre rien d'intéressant, ou bien bloquante si à l'inverse aucun fil d'exécution n'est prêt en local et que la seule possibilité d'avancer les calculs provient du réseau. Il est alors inutile de faire une scrutation active du réseau, il suffit de faire une attente passive, ce qui permet de partager le processeur avec d'autres applications. En conséquence une action importante est la scrutation (bloquante ou non) du réseau. Cette scrutation se fait de la façon suivante :

- d'abord un nouveau tampon de réception est défini (on ne doit pas écraser l'ancien tampon, qui sert peut être encore), ensuite une réception bloquante ou non, selon le choix préalablement fait, est effectuée,
- si un message est reçu, un descripteur de message lui est alloué et est partiellement rempli (identification de la tâche source...),
- le message est alors décrypté en fonction du service demandé :
    - le service demandé peut être un service système spécial (terminaison abrupte de la tâche, du programme...), il est alors exécuté immédiatement. Notons que pour effectuer cette exécution, il est possible d'appeler d'autres services sur d'autres tâches. C'est le cas par exemple pour la terminaison du programme : le service de terminaison est appelé sur les tâches filles et parente pour les terminer à leur tour.

- le service demandé peut être la réception d'un message de réponse. Le message de réponse contient l'identité d'un point de synchronisation qu'il suffit de signaler. Le message est raccroché au point de synchronisation pour pouvoir être déballé par le fil d'exécution qui l'attend.
- le service demandé peut être le service d'initialisation de la tâche, ou bien de terminaison. Auquel cas, l'état de la tâche évolue (selon le diagramme 4.4) et la fonction de service correspondante définie par l'utilisateur est lancée.
- Enfin, le service demandé peut être un service utilisateur. Si l'état de la tâche le permet (s'il vaut *INITIALIZED*) et que le service soit défini, alors la fonction de service correspondante est lancée. Sinon, une erreur fatale est produite.

Lors du lancement d'une fonction de service, soit la fonction peut être exécutée (la limite de concurrence pour le service n'est pas encore atteinte), soit elle est mise en attente dans une file et sera exécutée lorsque suffisamment de fonctions du même service seront terminées.

Lors de l'exécution d'une fonction de service, un nouveau fil d'exécution est créé, son contexte est mis à jour, en particulier, on y enregistre l'identité du message de requête, l'identité de l'appelant... ; un descripteur de message est aussi créé pour la réponse. Les fils d'exécution n'étant pas préemptifs, le fil nouvellement créé n'est pas exécuté immédiatement, mais seulement lorsque le contrôle revient dans l'opération d'ordonnancement-scrutation, qui décide, sachant qu'un nouveau fil vient d'être créé, de se bloquer au profit de celui-ci.

Le fil d'exécution qui vient d'être créé (ou réactivé, voir en 4.6.3.1), peut alors tourner. Il peut librement effectuer des opérations de calcul, envoyer des requêtes. Finalement, il va soit décider de se bloquer sur un sémaphore, en attente par exemple d'un résultat, soit terminer la fonction de service qu'il exécute.

S'il décide de se bloquer sur un sémaphore, il va simplement exécuter l'opération d'ordonnancement-scrutation.

S'il termine la fonction de service, il exécute un code qui consiste en :

- transmettre un éventuel message de réponse résultant de l'exécution du service,
- libérer les tampons des messages de requête et de réponse,
- changer éventuellement l'état de la tâche si c'est le service InitTask qui se termine (la tâche devient *INITIALIZED*),
- exécuter éventuellement une nouvelle requête pour ce service, qui était mise en attente à cause d'une limite de concurrence,
- ou bien exécuter l'opération d'ordonnancement-scrutation, en précisant que le fil d'exécution se termine.

Une petite optimisation est effectuée dans le cas de la terminaison d'un fil d'exécution. En effet, le fil désirant se terminer, qui a fait l'opération d'ordonnancement-scrutation et qui a mis à jour une nouvelle requête à exécuter, va directement l'exécuter, plutôt que de créer un autre fil d'exécution et lui donner la main en se suicidant. Cette petite optimisation économise une création et un changement de contexte, en particulier quand les services ne se bloquent pas (cas typique d'un ping-pong entre deux tâches).

Un problème se pose, pendant les opérations faites au cours de l'opération d'ordonnancement-scrutation, lorsqu'on doit « réveiller » *le* fil d'exécution, prétendument bloqué sur un sémaphore, qui effectue cette opération. En particulier, cela se produit lorsqu'un fil d'exécution qui veut se bloquer en attente d'un message, effectue une scrutation qui l'amène à découvrir le message qu'il attend. Il ne doit pas alors se « réveiller lui-même », puisqu'il n'est pas encore bloqué. Cette condition est détectée en testant le sémaphore que l'on signale vis à vis de celui sur lequel le fil d'exécution qui exécute

l'ordonnancement veut se bloquer. S'ils sont identiques, il ne faut pas signaler le sémaphore, mais mémoriser le fait que le fil d'exécution ne doit pas se bloquer à la fin de l'opération d'ordonnancement-scrutation.

Algorithmes d'ordonnancement-scrutation Les grandes lignes architecturales d'Athapascan-0a étant posées, nous allons maintenant décrire rapidement le cœur d'Athapascan-0a, c'est à dire l'algorithme d'ordonnancement- scrutation du réseau. Nous avons implanté deux modes. Le premier effectue la scrutation à chaque commutation des processus légers. Le second effectue la scrutation uniquement quand plus aucun processus léger n'est prêt à fonctionner. Nous aurions pu implanter aussi bien un mode intermédiaire qui effectuerai la scrutation tous les n ordonnancements.

Le premier mode a tendance à créer beaucoup de processus légers concurrents, développant l'arbre d'exécution en quelque sorte « en largeur », tandis que le second restreint le nombre de processus légers actifs en même temps au stricte minimum, développant l'arbre plutôt « en profondeur ». Notons qu'il est possible de forcer un ordonnancement particulier des calculs, ce qui est intéressant si l'ordonnancement est connu au préalable. Cependant dans le cas général, il n'y a pas d'ordonnancement connu statiquement et le fonctionnement est dynamique. Il est alors intéressant d'étudier des heuristiques simples de fonctionnement. Tout processus léger qui décide de se bloquer en attente d'un événement, ou bien de se suicider, doit d'abord exécuter une action d'ordonnancement. C'est cette action d'ordonnancement qui nous intéresse ici.

D'une façon générale, les messages de résultat sont considérés comme plus urgents que les messages de requête. Ils sont distingués les uns des autres à l'aide du mécanisme de *tags* de PVM. Cette considération permet de conserver un nombre de processus légers plus restreint, en terminant plus tôt les fils d'exécution qui attendent des réponses.

Voici maintenant l'algorithme du mode « scrutation quand aucun processus léger n'est prêt » :

```
A:
s'il existe un proc. léger prêt à tourner
    si suicide
        se mettre dans la file des proc. légers libres
        exécuter le harnais de point d'entrée (blocage et commutation);
        aller en A:
    sinon
        se bloquer (commutation); aller en B:
    fin si
sinon
    s'il y a un message de résultats disponible dans le réseau (scrutation)
        enregistrer le message dans le point de synchronisation associé
        s'il existe un proc. léger bloqué sur le point de synchronisation
            débloquer le proc. léger
            si suicide
                se mettre dans la file des proc. légers libres
                exécuter le harnais de point d'entrée (blocage et commutation);
                aller en A:
            sinon
                se bloquer (commutation); aller en B:
            fin si
        sinon
            si le proc. léger courant voulait se bloquer en attente de ce message
                aller en B:
            fin si
        fin si
    fin si
    s'il y a un message de requête disponible dans le réseau (scrutation)
        si suicide
            exécuter la requête;
            aller en A:
        sinon
            réquisitionner un proc. léger libre ou en créer un,
            lui transmettre la requête, le débloquer
            se bloquer (commutation);
            aller en B:
        fin si
    fin si
    attente bloquante sur le réseau
    aller en A:
fin si
B:
```

FIG. 4.24 - *Algorithme de scrutation « quand aucun procesus léger n'est prêt ».*

Dans ce mode, si un autre processus léger est prêt, il y a commutation. Dans le cas contraire, il y a scrutation des messages présents sur le réseau. Chaque message est soit un message de résultat, soit un message de requête. Si c'est un message de résultat, le point de synchronisation auquel il est destiné sera modifié pour garder sa trace. Le cas spécial où le message de résultat est celui attendu par le processus léger qui fait l'opération d'ordonnancement est traité à part. Si le message est un message de requête, deux cas sont possibles : soit le processus léger qui exécute cet ordonnancement voulait se suicider, et il exécute directement la requête reçue. Soit il voulait seulement se bloquer, et il réquisitionne un processus léger libre ou bien en crée un nouveau, lui donne la requête et commute

vers lui.

L'algorithme de scrutation à chaque commutation est semblable ; nous ne le décrirons pas ici.

4.6.3.5 Initialisation / terminaison d'Athapascan-0a

En partie pour des raisons historiques (la première implantation ayant été faite sur le noyau de processus légers « Briat's Core », qui imposait cette structure), mais aussi parce que l'API d'Athapascan-0a l'oblige, le *main()* n'est pas visible du programmeur. La fonction *main()* se trouve dans le module d'interface aux noyaux de processus légers, puisque certains noyaux (Briat's Core en particulier) la définissent. Cette fonction *main()* contient principalement l'initialisation du noyau de processus légers, puis l'appel à une fonction nommée *ADAM()*.

La fonction *ADAM()*, début de l'exécutif proprement dit, évalue la ligne de commande, initialise certaines tables, démarre le démon PVM si besoin (s'il n'est pas déjà présent), puis effectue soit l'initialisation de la tâche racine, soit l'initialisation d'une tâche ordinaire. La sélection entre les deux options est faite en fonction des arguments présents sur la ligne de commande (une tâche ordinaire possède obligatoirement un argument spécial, qui l'informe de l'identification de son créateur). L'initialisation de la tâche racine se réduit à définir un pseudo-service qui devra être appelé au préalable : il s'agit de la section programme définie par l'utilisateur dans la tâche racine. L'initialisation d'une tâche ordinaire consiste en une lecture des arguments spécifiques sur la ligne de commande (identification de la tâche parent, identification de la tâche gestionnaire d'environnement...), la déclaration des services de base (InitTask et TermTask), puis à répondre au parent que la tâche est chargée.

La section programme contient à son tout début la déclaration des services de l'environnement que devra servir la tâche racine. Après cette déclaration, la tâche racine passe directement à l'état *INITIALIZED*, sans nécessiter l'exécution du point d'entrée *InitTask*.

A la fin de l'initialisation, que ce soit de la tâche racine ou d'une tâche ordinaire, le code particulier, dit « chien de garde » est exécuté. Ce code commence par se déclarer comme étant l'exécution d'un service spécial, en particulier pour que le fil d'exécution correspondant puisse être tracé correctement. Dans l'implantation initiale, le chien de garde est un fil d'exécution particulier qui se bloque pour laisser tourner les fils de calcul puis est réactivé et effectue l'ordonnancement-scrutation et la détection de terminaison de la tâche. Dans l'implantation actuelle, optimisée, la fonctionnalité du chien de garde est directement effectuée par les fils de calcul ; le chien de garde existe toujours mais ne fait plus grand chose : si l'on est dans la tâche racine, il active la section programme comme un fil d'exécution séparé (la section programme de la tâche racine correspond au service InitTask d'une tâche ordinaire). Il effectue ensuite une opération d'ordonnancement-scrutation, en se bloquant sur un sémaphore qui lui est dédié. Il reste bloqué durant toute l'exécution du programme, l'ordonnancement-scrutation et la détection de terminaison étant maintenant effectuées par les fils de calcul eux-mêmes. Lors de la terminaison de la tâche, le chien de garde est signalé. Il sort donc de son blocage sur son sémaphore, puis déconnecte la tâche de PVM et demande la terminaison du processus à son noyau de multiprogrammation.

Lorsqu'une tâche provoque une erreur d'exécution fatale, l'arborescence de création des tâches est parcourue pour propager l'arrêt de toutes les tâches. L'utilisateur n'est donc pas laissé avec un état intermédiaire du système ; il reçoit en plus un message précisant la tâche d'origine de l'erreur.

4.6.3.6 Fonctionnalités annexes de l'implantation

Nous décrivons ici quelques fonctionnalités annexes de l'implantation d'Athapascan-0a.

L'environnement L'environnement est implanté d'une façon très simple, comme un service centralisé. Ce service possède deux variantes, l'une pour lire la valeur d'une variable, l'autre pour l'écrire. Les fils d'exécution n'étant pas préemptibles, il n'est pas nécessaire de verrouiller l'exécution concurrente de ce service. L'environnement est géré par la tâche racine. Actuellement, les fonctions de l'environnement sont une simple redirection sur la gestion de l'environnement (du *shell*) de la tâche racine. Cela permet par exemple de configurer l'exécution d'un programme Athapascan-0a à l'aide de variables d'environnement au niveau du shell. Notons que l'environnement peut être dynamique, mais que la gestion de sa cohérence éventuelle est à la charge du programmeur. De plus, étant donné son implantation très inefficace, c'est un simple outil de diffusion d'informations stables, et cela ne peut pas être considéré comme une mémoire partagée d'usage général. Cependant, son utilité n'est plus à démontrer.

La tâche console texte La tâche console texte est implantée comme une tâche séparée, qui s'exécute dans une fenêtre de *xterm*. Les deux services *READ* et *WRITE* sont implantés d'une façon triviale. Le nom de la tâche console peut être diffusé par l'intermédiaire de l'environnement.

La tâche console graphique La tâche console graphique est implantée comme une tâche séparée, qui s'exécute dans plusieurs fenêtres graphiques. Cette tâche propose un certain nombre de points d'entrée pour :

- créer une nouvelle fenêtre,
- fermer une fenêtre,
- associer une palette de couleurs à une fenêtre,
- dessiner des points, des lignes, des carrés, des cercles,
- afficher un *bitmap*,
- écrire du texte.

Ces différentes fonctionnalités sont une simple interface à des appels au serveur X11 qui gère les fenêtres. Ici encore, puisque les fils d'exécution ne sont pas préemptibles, il est inutile de verrouiller la concurrence des actions.

Options de compilation L'exécutif Athapascan-0a peut être compilé pour une machine homogène ou hétérogène. L'exécutif Athapascan-0a peut aussi être compilé pour fournir diverses instrumentations de l'exécution. Enfin, Athapascan-0a peut inclure les mécanismes de réexécution déterministe ou bien les inhiber ; dans ce dernier cas, la taille des entêtes des messages est réduite à seulement deux entiers.

Options d'exécution Les options d'exécution des applications Athapascan-0a concernent la prise de trace pour la réexécution déterministe, l'efficacité accrue par l'utilisation des modes *data-inplace* et *direct-route*, avec certaines restrictions sur la programmation cependant, l'option *secure* qui permet de vérifier le type des données échangées, une option *greedy* permettant de changer l'ordonnancement des activités vis à vis des requêtes, et enfin, un grand nombre de classes de déverminage, permettant de tracer l'exécution de telle ou telle partie de l'exécutif ou du programme.

Déverminage Un programme Athapascan-0a peut être déverminé comme un programme PVM, à l'aide de dévermineurs lancés dans des fenêtres de *xterm*. L'exécutif Athapascan-0a autorise un mode de déverminage dans lequel toute tâche lancée par le programme est déverminée. Il est aussi possible de raccrocher un dévermineur manuellement à une tâche particulière. Notons que pour correctement

déverminer un programme Athapascan-0a, il est préférable que le dévermineur soit au courant de l'existence de processus légers à l'intérieur du processus. Cependant, puisque qu'Athapascan-0a utilise des fils d'exécution non préemptibles, il est relativement aisé d'utiliser un dévermineur de base comme *dbx*, même s'il n'est pas au courant de l'existence des fils d'exécution. En particulier, il n'y a pas à craindre un effet de bord caché dû à l'exécution en concurrence d'autres fils d'exécution durant le déverminage d'une procédure. Le problème principal se résume au fait que l'on doit attraper le « bon » fil d'exécution, et qu'il est difficile de forcer une commutation pour examiner un autre fil d'exécution.

Considérations pour l'implantation de l'API Toutes les fonctions de l'API retournent un code d'erreur. Toutes les structures manipulées par l'utilisateur contiennent un *checksum* permettant de vérifier que la structure a bien été forgée par l'API. Ce point est très important, puisqu'il arrive souvent que les utilisateurs novices oublient d'initialiser, référencent incorrectement, ou bien forgent par erreur ces structures.

4.6.4 Conclusion sur l'implantation

L'implantation d'Athapascan-0a repose donc fortement sur PVM, pour l'administration de la machine virtuelle, des processeurs virtuels, la réalisation des communications point à point et la gestion de l'hétérogénéité de format de données. Différents noyaux de multiprogrammation, coopératifs, ont simplement été accolés à PVM ; le code source de PVM n'a pas été modifié. Nous avons vu que le contrôle de flux, du fait des spécificités de PVM, reste à la charge d'une couche de régulation ou de l'utilisateur. Nous avons décrit le déroulement d'un appel de service, en fonction des opérations PVM sous-jacentes ; les structures de données principalement utilisées ; le cycle d'exécution d'un service. Celui-ci repose sur une phase d'ordonnancement-scrutation visant à faire progresser calculs et communications ; nous avons décrit en détail un algorithme d'ordonnancement-scrutation particulier, qui n'effectue la scrutation du réseau que quand aucun processus léger n'est prêt à tourner. Les grandes lignes de l'initialisation et de la terminaison d'un programme Athapascan-0a ont aussi été données. Enfin, l'implantation de différentes fonctionnalités annexes (environnement global, tâches consoles...) a été abordée. Nous allons maintenant conclure cette description de l'implantation d'Athapascan-0a par une récapitulation des portages effectués et une discussion sur la portabilité de cette implantation sur un autre noyau de communication que PVM.

4.6.4.1 Récapitulation des portages

Différents noyaux de processus légers ont été utilisés pour les différents portages d'Athapascan-0a. Grâce à la portabilité de PVM, tous les portages d'Athapascan-0a sont inter-opérables. Porter Athapascan-0a d'une machine à une autre est très simple : comme PVM est largement répandu, seul un noyau de processus léger de base de niveau utilisateur est nécessaire.

Machine	Processeur	Système	Noyau de multiprogrammation	Noyau de communication
Sun	M68K, Sparc	SunOS 4.1.3	Sun LWP, REX	PVM
Sun	Sparc	Solaris 2.5	Solaris LWP	PVM
DG Aviion	M88K	DG-Unix	Briat's Core	PVM
PC	ix86	Linux 1.3	REX, pthreads Mueller	PVM
Dec Farm	Alpha	OSF/1 3.0	DCE Threads	PVM
IBM SP	RS6K	Aix 3.2.5	DCE Threads	PVM, PVMe

TAB. 4.2 - *Les portages d'Athapascan-0a.*

4.6.4.2 Portabilité sur un autre noyau de communication que PVM

PVM présente, globalement, deux fonctionnalités majeures : la gestion d'une machine virtuelle et les communications point à point par échanges de messages. Beaucoup de noyaux de communications ne présentent pas de gestion de machine virtuelle (c'est ce qui a fait la force de PVM, et qui continue d'ailleurs). Cependant, les fonctionnalités requises par Athapascan-0a peuvent être réécrites spécifiquement ou restreintes (par exemple en utilisant toujours une machine virtuelle statiquement définie). Les communications de PVM sont aussi spécifiques dans le sens où la gestion des tampons de communication échappe au programmeur. C'est à la fois un bien (plus de simplicité) et un mal (moins d'efficacité). Tout portage d'Athapascan-0a sur un autre noyau de communication doit, puisqu'il dépend fortement du modèle de tampons infinis de PVM, reconstruire la mécanique de tamponnage de PVM. Cette mécanique n'est pas très efficace mais semble particulièrement adaptée aux applications parallèles irrégulières, pour lesquelles les données sont souvent stockées de façon imprévisible en mémoire. L'utilisation d'une carte mémoire, relativement statique, comme celle employée par MPI, semble inappropriée dans ce cadre.

4.7 Analyse des performances d'Athapascan-0a

Nous analysons ici le surcoût introduit par l'utilisation d'Athapascan-0a comparé à l'usage direct de PVM ou PVMe. Nous commençons par présenter l'intérêt des diverses optimisations qui ont été menées entre les premières versions d'Athapascan-0a et la version finale. Nous évaluons ensuite le coût de la multiprogrammation et le coût de la scrutation du réseau. Ces mesures sont très proches de l'implantation. D'une façon plus générale, nous comparons les critères de base de tout système de communication : la latence et le débit natifs et ceux obtenus avec Athapascan-0a. Le but principal d'Athapascan étant de permettre un recouvrement (automatique) des attentes des communications par du calcul, nous évaluons aussi ce point. Enfin, nous montrons sur une véritable application de calcul formel, qu'Athapascan-0a peut être très compétitif vis à vis de son support natif.

4.7.1 Optimisations diverses

Les optimisations apportées aux premières versions du prototype sont de quatre ordres :

- la suppression des allocations dynamiques pour la plupart des structures de données importantes ; les structures sont maintenant allouées par blocs, avec une gestion en liste de libres des structures libérées, permettant ainsi une allocation quasi-immédiate et un coût des gestion minimal. Un léger inconvénient de cette organisation est que la mémoire allouée n'est jamais réellement libérée, en prévision d'une utilisation future,

- la réduction de l'entête des messages ; cet entête est maintenant de deux octets seulement (contre plus d'une dizaine auparavant),
- la réécriture de l'opération d'ordonnancement-scrutation. Dans la première version, la scrutation est toujours faite par le « chien de garde », le fil d'exécution qui contrôle la tâche. Ce fil d'exécution fonctionne donc comme un démon de communication. L'inconvénient est que chaque réception de message nécessite en général deux commutations de fils d'exécution, du fil d'exécution qui se bloque vers le chien de garde et de celui-ci vers le fil qui est démarré ou réveillé. Dans la version optimisée, l'opération d'ordonnancement-scrutation est directement exécutée par le fil d'exécution qui se bloque ou se termine. Cette optimisation permet donc de ne faire qu'une commutation (du fil d'exécution se bloquant vers celui qui est créé ou réveillé par la réception d'un message), voire aucune dans le cas où le fil d'exécution qui souhaite se bloquer s'aperçoit que le message qu'il attend est déjà arrivé, ou bien dans le cas où le fil d'exécution se termine et peut directement exécuter la requête suivante).
- la réutilisation des fils d'exécution. Créer un fil d'exécution a un coût proche d'une simple commutation dans la version optimisée.

Nous avons mesuré une version préliminaire et la version optimisée d'Athapascan-0a. Nous donnons dans la suite une comparaison de ces deux versions, pour indiquer quels gains de performance peuvent être atteints par ces optimisations relativement simples.

4.7.2 Coût de la multiprogrammation

Le coût de la multiprogrammation peut être évalué par deux points caractéristiques : le coût d'allocation / libération d'un fil d'exécution et le coût de commutation entre deux fils d'exécution. Nous ne mesurons pas les performances natives des noyaux de multiprogrammation, uniquement celles répercutées à travers l'interface d'accès utilisée par Athapascan-0a. Dans ces tests, tous les programmes sont optimisés.

4.7.2.1 Coût d'allocation / libération d'un fil d'exécution

Le coût d'allocation / libération d'un fil d'exécution est important puisque, à priori, tout appel de procédure à distance est réalisé par Athapascan-0a comme l'allocation d'un nouveau fil, puis sa libération à la fin de l'exécution de la fonction du service. L'optimisation de réutilisation des fils d'exécution permet de réduire ce coût et de le rendre proche d'un double coût de commutation. Toutefois cette optimisation est fortement reliée au fait que toutes les piles sont de même tailles. Dans le but de distinguer le coût initial de création / destruction (avec allocation et libération de la pile) du coût de réutilisation (qui s'apparente à une double commutation, une vers un fil d'exécution libre à l'allocation, une autre vers un fil d'exécution prêt à la libération), nous faisons deux tests :

Le premier crée un nombre important de fils d'exécution. Cette phase se fait sans préemption du fil qui l'exécute. Celui-ci se bloque ensuite sur un sémaphore, pour donner la main à ceux qu'il vient de créer. Ceux-ci se suicident tour à tour. Le dernier réveille le créateur. Nous mesurons la séquence complète de créations, commutations et destructions.

Le second test consiste à pré-allouer les fils d'exécution. Ceci est simulé simplement en exécutant deux fois la séquence du premier test. La mesure n'est faite qu'à la seconde exécution ; de ce fait dans la version optimisée, on mesure le coût de la réutilisation d'un fil d'exécution inactif, plutôt que celui de sa création initiale. Dans la version non optimisée, on évalue si le noyau de multiprogrammation effectue lui-même une optimisation lors de la réutilisation

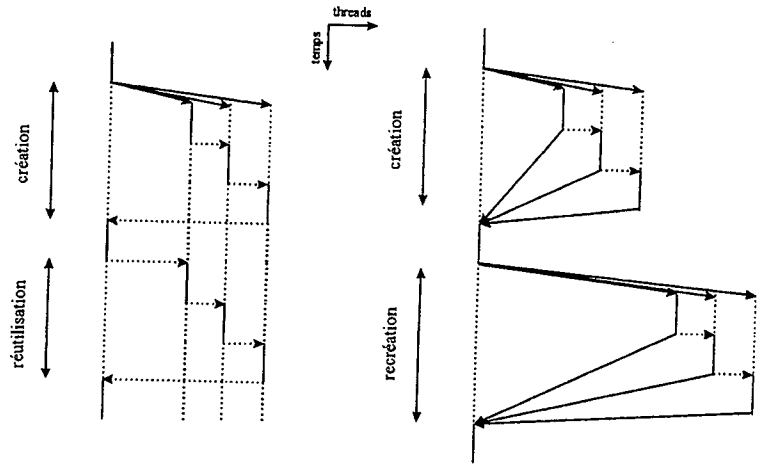

FIG. 4.25 - *Test création / recréation / réutilisation.*

A gauche, trois processus sont créés, exécutés, et réutilisés dans une deuxième phase. A droite, trois processus sont créés puis détruits et trois nouveaux processus créés.

Machine	création initiale avec ou sans optimisation	réutilisation avec optimisation	réutilisation sans optimisation
Pentium 100 Mhz cache pipeline, Linux, REX	170 μs	7.5 μs	21 μs
Nœud du SP1 (Power-1 à 60Mhz), Aix, DCE threads	510 μs	105 μs	220 μs
Nœud de la ferme d'Aphas (DECchip 21064 à 133Mhz), OSF/1, DCE threads	1450 μs	210 μs	variable de 2300 à 7800 μs
Navajo (SuperSparc 60Mhz), Solaris, threads Solaris	1270 μs	(pas d'optimisation programmée)	1270 μs

TAB. 4.3 - *Temps de création, commutation et destruction d'un fil d'exécution.*

Lors d'une création initiale ou lors d'une réutilisation, avec ou sans optimisation.

- Lors d'une création initiale, les actions effectuées sont différentes selon la version :
  - dans la version non optimisée, il y a la création initiale du fil d'exécution, une commutation vers lui, son exécution et son suicide (qui inclut forcément une commutation vers un autre fil),
  - dans la version optimisée, il y a la création initiale du fil d'exécution, une commutation vers lui, l'établissement d'un point de saut non-local, l'exécution de la fonction demandée, un saut non local pour terminer l'exécution en cours et une commutation vers un autre fil prêt.

Cependant, ces différentes actions doivent se compenser, car l'optimisation n'influe pas sensiblement sur le temps de création initiale.

Lors de la réutilisation avec optimisation, il y a au moins une commutation vers le fil d'exécution pré-créé en attente et une autre lors de la destruction. Le coût de réutilisation est donc au minimum le double de celui de commutation seule. Il y a d'autre part l'établissement d'un point de saut non local et un saut non local. Sur les Pentiums et les Alphas, ces opérations semblent très rapides, alors que sur les Powers, cela grève sensiblement le temps de réutilisation.

Dans la version sans optimisation, on voit que la « réutilisation » (en fait une nouvelle création) coûte le même coût qu'une création initiale pour Solaris. Elle coûte très cher sur OSF/1 (peut être à

cause du ménage qu'il est nécessaire de faire entre les créations successives). Notons que ce coût est très variable, preuve que l'interaction avec le système est importante. Sur le SP1 et les Pentium, la « réutilisation » sans optimisation coûte plus cher qu'avec optimisation, mais moins cependant que la création initiale. Dans les deux cas, la ré- allocation de la pile est peut être accélérée par l'allocateur de mémoire sous-jacent puisque la taille des blocs demandés est identique à chaque fois. En particulier, REX ne fait pas d'optimisation de réexécution, l'accélération par rapport à la création initiale est donc imputable au système.

En conclusion, notons que le temps de création initiale peut être assez important, mais que le temps de réutilisation avec l'optimisation est plus raisonnable et permet d'envisager un grain de parallélisme fin. Notons que le noyau REX, dont les fonctionnalités sont très limitées, est très rapide.

4.7.2.2 Coût de commutation

Le coût de commutation est évalué en faisant une très grande séquence de commutations entre deux fils d'exécution. Chacun réveille l'autre et se bloque tour à tour.

Machine	Athapascan-0a avec optimisation	Athapascan-0a sans optimisation
Pentium 100 Mhz cache pipeline, Linux,REX	3.5 μs	2.6 μs
Nœud du SP1 (Power-1 à 60Mhz), Aix, DCE threads	24 μs	16 μs
Nœud de la ferme d'Aphas (DECchip 21064 à 133Mhz), OSF/1, DCE threads	100 μs	950 μs
Navajo (SuperSparc 60Mhz), Solaris, threads Solaris	(pas d'optimisation programmée)	42 μs

TAB. 4.4 - *Temps de commutation.*

Avec ou sans optimisation de réutilisation

Sur les Pentium, le léger surcoût apparent de commutation est dû à une lecture de la mémoire privée du fil d'exécution pour reconnaître son identité pour Athapascan-0a. Cette lecture est effectuée en dehors du chemin de mesure pour la version non optimisée, mais en toute généralité doit être faite quand même. Sur le SP1 et les Alphas, l'algorithme de commutation a changé entre la version non optimisée et la version optimisée. Dans la version non optimisée, un yield() est effectué pour la commutation. Cette fonction semble très lente sur les Alphas (probablement parce qu'elle cause une interaction avec le système OSF), mais est plus rapide, sur le SP1, que la séquence à base de variable de condition employée dans la version optimisée. Cette plus grande rapidité sur le SP1 n'est probablement qu'apparente : dans notre test, un seul fil d'exécution est bloqué sur un sémaphore. Si plusieurs fils sont bloqués (cas le plus courant dans un programme réel), dans la version non optimisée ils sont tous réveillés tour à tour jusqu'à trouver celui qui peut s'exécuter, alors que dans la version optimisée, le fil qui peut s'exécuter est directement remis dans la file des prêts.

Le coût de commutation est une caractéristique importante d'un noyau de multiprogrammation ; nous voyons, comme dans le cas des Alphas, qu'un choix judicieux d'une technique particulière peut aboutir à des améliorations très importantes. Cependant, le choix fait pour un noyau de multiprogrammation donné n'est pas forcément applicable aux autres.

En conclusion, le coût de commutation associé aux processus légers est très faible et permet bien un grain de parallélisme fin.

4.7.3 Coût de la scrutation

Le coût de scrutation est un autre facteur important, puisque la scrutation peut être faite très souvent, en fonction de l'algorithme d'ordonnancement. Nous évaluons seulement le coût d'une scrutation non bloquante qui échoue à retourner un message. Ce cas de figure est le plus courant. Si la scrutation retourne un message le temps de traitement du message peut rentabiliser la scrutation ; le cas que nous mesurons est donc en quelque sorte le « pire » puisque la scrutation est faite pour rien.

Machine	durée de pvm_nrecv(-1,-1)
Pentium, Linux, PVM sur IP/Ethernet	62 μs
Power, Aix, PVM sur IP/Ethernet ou IP/HPS	71 μs
Power, Aix, PVMe sur HPS	0.64 μs
Alpha, OSF/1, PVM sur IP/Ethernet ou IP/GigaSwitch	976 μs
Sparc, Solaris, PVM sur IP/Ethernet	2.3 μs

TAB. 4.5 - *Temps de scrutation.*

Non bloquante, sans réception réelle

La scrutation de PVMe est très rapide ; c'est probablement un simple test en mémoire. La scrutation de PVM sur Solaris est aussi assez rapide. Notons que la version de PVM utilisée par Solaris est la version à mémoire partagée ; cela explique peut être la différence avec les autres mesures de PVM réalisées sur les Pentiums ou les Powers, qui forment bloc. Enfin, la scrutation sur Alphas est très lente, peut être à cause d'une interaction avec le système OSF/1. Notons que dans les deux cas, PVM sur IP/Ethernet ou PVM sur IP/réseau rapide, la scrutation coûte le même temps. Cela signifie que le réseau n'est pas réellement accédé ; seule l'interaction avec les couches basses du système est mesurée.

Notons que l'utilisation du mode *direct-route* ou d'un filtrage à la scrutation (scrutation des messages présentant une étiquette particulière) n'influence en rien ces valeurs.

Le coût de la scrutation du réseau est très variable d'un système à l'autre ; en conséquence, l'algorithme d'ordonnancement-scrutation optimal varie lui aussi. Par exemple avec PVMe, une scrutation à chaque commutation semble tout à fait acceptable ; par contre sous OSF/1, limiter la scrutation au moments où aucun fil d'exécution n'est prêt semble préférable. Nous n'avons pas effectué d'expérience pour confirmer cette impression.

4.7.4 Comparaison de performances entre PVM et Athapascan-0a : latence et débit maximum

Deux critères importants pour juger de l'efficacité d'un mécanisme de communication sont la latence et le débit. La latence est le « temps d'initialisation d'un message ». Plus précisément, nous la définirons ici comme la moitié du temps nécessaire à l'envoi d'un message de taille minimum et à la réception d'une réponse de même taille. Envoi et réception peuvent avoir des durées différentes, mais pour ce qui nous intéresse ici, nous considérerons la valeur moyenne seulement. Le débit sera défini comme la quantité de données pouvant être échangées dans un intervalle de temps. Le débit varie en fonction de la taille des messages échangés.

Comme nous l'avons vu dans l'introduction, il est important de pouvoir modéliser le comportement d'une application. Couramment, une approximation de la courbe de durée de communication d'un message en fonction de sa taille peut être donnée par une droite, dont le point d'origine est la

latence théorique et dont la pente est donnée par l'inverse de la « bande passante ». La latence théorique n'est pas identique à la durée de communication d'un message de taille minimum (la latence expérimentale), car la courbe de durée est seulement approchée par une droite. De même, la bande passante ne représente pas le débit maximum, seulement la valeur asymptotique du débit. Notons sur la courbe de débit en fonction de la taille des messages un point important : celui du demi débit maximum. Cette valeur indique à partir de quelle taille de message le réseau de communication est utilisé de façon « efficace ». Si l'on communique à l'aide de messages trop petits, seule une fraction du débit maximum sera atteinte. En revanche si l'on communique à l'aide de messages dont la taille est plus grande que celle nécessaire au demi débit maximum, une part importante du débit maximum sera atteinte. La taille des messages pour le demi débit maximum est donc un critère important pour le concepteur d'une application. Notons que le choix du demi débit maximum est arbitraire, nous pourrions considérer de même les 80% du débit maximum ou tout autre fraction.

Athapascan-0a ayant été implanté au-dessus de différentes versions de PVM, une question intéressante est de savoir quelle fraction de performances est perdue par Athapascan-0a par rapport à PVM. Pour obtenir la réponse à cette question, nous avons testé un petit programme en PVM et en Athapascan-0a, programme qui effectue un *ping-pong* entre deux tâches. La tâche cliente prépare un message, emballe une certaine quantité de données, envoie le message au serveur. La tâche serveur reçoit le message, déballe les données, détruit le message, prépare une réponse, emballe la même quantité de données que celle qu'elle a reçue et envoie la réponse. La tâche client reçoit la réponse, déballe les données et détruit la réponse. Notons que cet exemple modélise de façon simpliste un cycle complet d'appel client-serveur, contrairement à beaucoup de tests de performances sur les noyaux de communication, qui par exemple ne font pas la phase emballage / déballage des données. Ce cycle est exécuté un nombre de fois suffisant pour obtenir des valeurs stables. Le programme PVM et le programme Athapascan-0a sont identiques du point de vue sémantique ; ils sont aussi similaires du point de vue de l'allocation des ressources. Les deux programmes sont optimisés et sont mesurés de la même façon. En conséquence, le surcoût présenté par Athapascan-0a est dû à la multiprogrammation des communications.

Nous avons fait tourner cet exemple sur quatre architectures différentes :

- sur un réseau local Ethernet, avec PVM domaine public sur IP, entre les nœuds du IBM SP,
- sur un réseau rapide « GigaSwitch », avec PVM domaine public sur IP, entre les nœuds d'une ferme d'Alpha,
- sur le réseau rapide « HighPerformanceSwitch », avec PVM domaine public sur IP, entre les nœuds du IBM SP,
- sur le réseau rapide HPS, avec PVMe (sur le mode *user-space* du HPS), entre les nœuds du IBM SP.

Notons que notre IBM SP est une version mixte entre le SP1 et le SP2 (parfois appelée SP1.5), c'est à dire avec des processeurs Power1 et des adaptateurs HPS TB2.

Les résultats obtenus sont représentés dans les courbes suivantes :

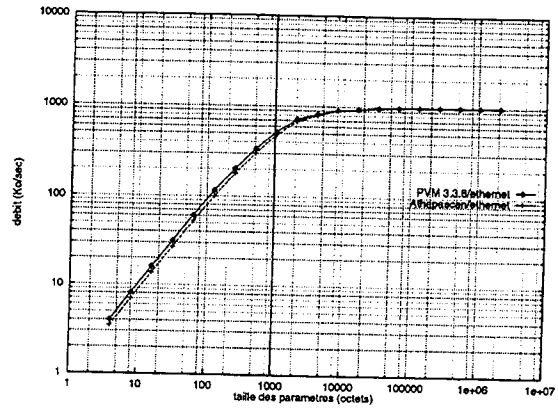

FIG. 4.26 - *Courbes des débits comparés d'Athapascan-0a et de PVM. Sur IP/Ethernet*

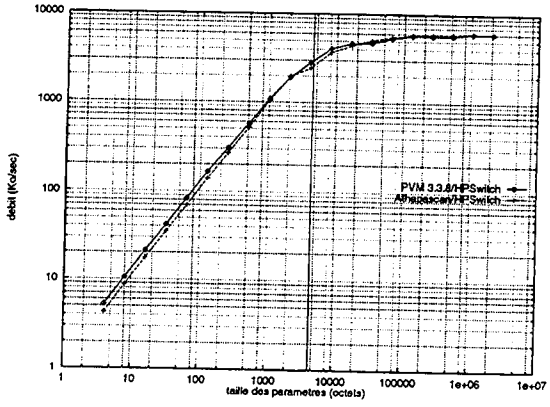

FIG. 4.27 - *Courbes des débits comparés d'Athapascan-0a et de PVM. Sur IP/HPS*

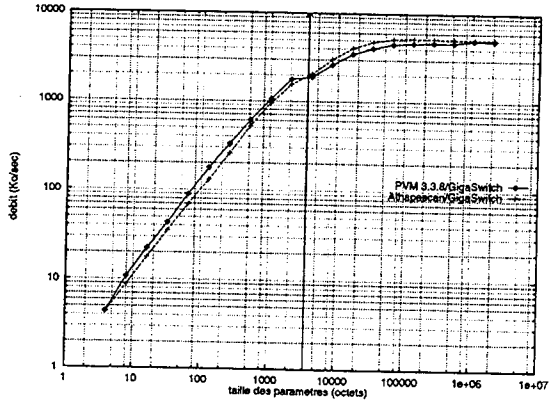

FIG. 4.28 - *Courbes des débits comparés d'Athapascan-0a et de PVM. Sur IP/GigaSwitch*

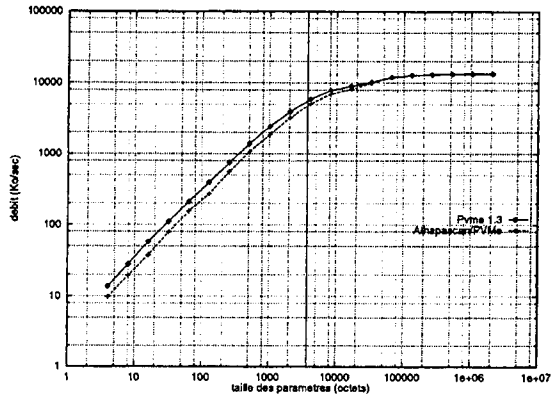

FIG. 4.29 - *Courbes des débits comparés d'Athapascan-0a et de PVMe. Sur CSS/HPS*

Ces courbes ont une échelle logarithmique sur les deux axes. Donc, même une petite différence compte. Nous remarquons que pour les petits messages, un surcoût important existe pour Athapascan-0a. Ce surcoût est mieux visible si l'on trace les petits messages seulement :

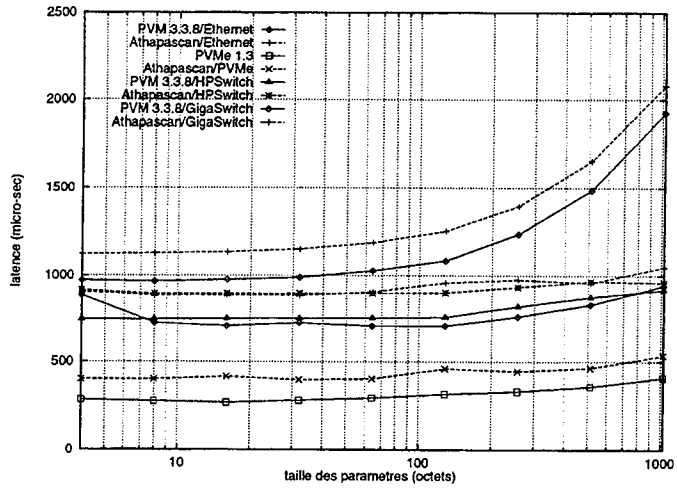

FIG. 4.30 - *Latences comparées d'Athapascan-0a et de PVM.*

On remarque que la valeur du surcoût peut atteindre 50% dans le cas de PVMe, mais seulement 15% dans celui d'un réseau Ethernet habituel. Ces valeurs ne sont pas très bonnes, mais elles concernent des petits messages. Si l'on regarde le surcoût pour des messages de taille raisonnable, proches ou plus grands que la taille de messages nécessaire pour atteindre le demi débit maximum[2], on s'aperçoit que dans tous les cas, *le surcoût est inférieur à 15%*. Il est même inférieur à 5% pour l'expérience sur Ethernet standard. En conséquence, pour une application de granularité de communication adaptée à la machine cible, Athapascan-0a n'induit pas un surcoût très important. On remarque d'autre part que le débit maximal d'Athapascan-0a et de la communication native sont identiques. Cela est du au fait qu'Athapascan-0a ne duplique pas les messages. Le surcoût est donc constant, lié au traitement effectué à l'arrivé du message, et non proportionnel à la taille des messages. Enfin, notons que sur le réseau Ethernet le surcoût relatif est plus faible : le réseau est plus lent, alors que le coût de traitement du message reste constant (les processeurs restent les mêmes).

2. Cette taille est indiquée par un axe vertical sur les courbes

Si l'on regarde la différence entre la durée de communication en PVM et en Athapascan-0a, on s'aperçoit qu'elle est quasiment constante quelle que soit la taille des messages, ce à quoi l'on s'attendait :

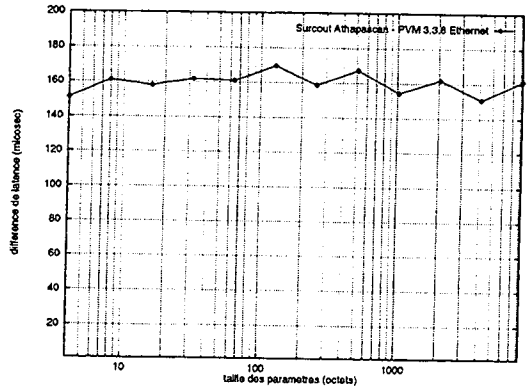

FIG. 4.31 - *Surcoût d'Athapascan-0a sur PVM sur IP/Ethernet.*

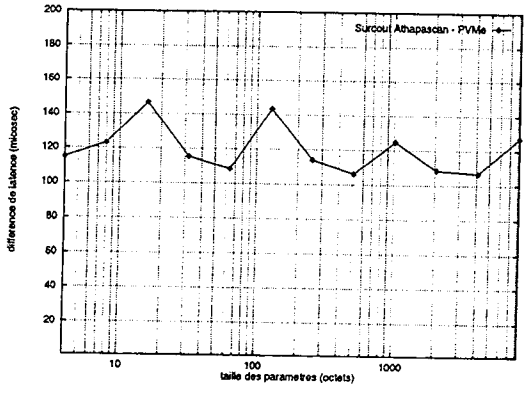

FIG. 4.32 - *Surcoût d'Athapascan-0a sur PVMe sur CSS/HPS.*

Cette valeur varie un peu en fonction de l'architecture utilisée puisque l'efficacité des noyaux de communication diffère.

En résumé nous donnons un tableau des performances comparées d'Athapascan-0a et de PVM (les débits maximaux d'Athapascan-0a et de PVM sont identiques) :

Architecture	latence PVM	latence Athapascan- 0a	surcoût	débit maximal
PVM 3.3.8 Ethernet	970μs	1120μs	150μs	1.0Mo/s
PVM 3.3.8 HighPerfSwitch	740μs	890μs	150μs	6.0Mo/s
PVM 3.3.8 GigaSwitch	690μs	890μs	200μs	5.2Mo/s
PVMe 1.3 (HPS)	270μs	400μs	130μs	13.7Mo/s

TAB. 4.6 - *Récapitulatif des performances comparées d'Athapascan-0a et de PVM.*

En conclusion, nous voyons que malgré l'introduction de la multiprogrammation, le surcoût d'Athapascan-0a par rapport aux communications natives reste très acceptable.

4.7.5 Efficacité de la multiprogrammation légère dans Athapascan-0a

Pour connaître l'efficacité de la multiprogrammation légère pour le recouvrement des attentes de résultats par d'autres calculs, nous avons utilisé le même algorithme de ping-pong entre deux tâches que précédemment, mais modifié comme suit. D'abord, le service appelé n'est plus « vide », en plus de la copie des paramètres reçus dans son tampon de résultats, il effectue un calcul synthétique représentée par une boucle vide d'un certain nombre d'itérations. D'autre part, l'appelant n'effectue pas seulement l'appel du point d'entrée distant, mais effectue le même calcul, après complétion de l'appel. Enfin, le nombre de processus légers dans chaque tâche n'est plus de un, mais est un paramètre de l'expérience, comme le sont la charge de calcul (nombre de cycles) et la taille des messages échangés. Chaque processus léger fait plusieurs cycles {appel, calcul} en séquence. Quand un processus léger se bloque en attente des résultats de l'exécution distante, un autre peut prendre la main. Nous obtenons donc un recouvrement des attentes de communication par du calcul. Dans la mesure où la charge de calcul exécutée à chaque appel par le serveur et par le client sont identiques, le facteur maximum de recouvrement est de deux, qui correspond au cas où un calcul est toujours effectué en parallèle sur le serveur et le client, alors que dans le cas où l'on ne disposait que d'un processus léger par tâche, les calculs serveur et client devaient être faits en séquence. Toutes les courbes sont présentées en normalisant le temps pour n processus légers sur le temps pour un.

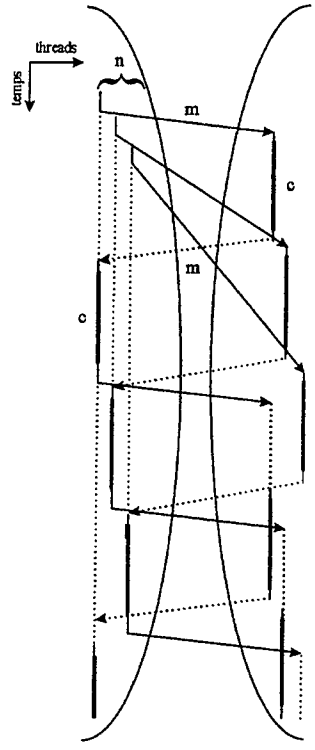

FIG. 4.33 - *ping-pong multiple.*

Les paramètres de l'expérience sont n, le nombre de fils d'exécution, c, le temps de calcul et m, la taille des messages échangés. Le ping-pong intervient entre deux tâches.

Cette expérience a été réalisée sur le IBM SP1, avec le réseau HPS. Mille cycles durent à peu près 40μs, soit une bonne partie de la latence de communication. En toute généralité nous devrions présenter des graphiques 4D, avec comme axes le nombre de processus légers, le nombre de cycles, la taille des messages et le rapport entre le temps pour n processus légers et le temps pour un. Comme la visualisation d'un graphique 4D est difficile, nous allons examiner des tranches en deux dimensions.

Si l'on fixe la charge de calcul (le nombre de cycles), on s'aperçoit que pour une grande charge de calcul (40ms, fig. 4.36), le recouvrement est maximal dès que l'on utilise deux processus légers. Pour une charge de calcul inexistante (fig. 4.34), le recouvrement est plus faible, puisqu'alors on ne recouvre l'attente de résultat que par la préparation d'autres communications. Dans la mesure où cette préparation est moins coûteuse que l'attente, on peut introduire un grand nombre de processus légers pour accroître le recouvrement, qui ne peut pas être complet cependant. Pour une charge de calcul moyenne (fig. 4.35), le recouvrement est réalisé avec un petit nombre de processus légers, mais quand même supérieur à deux. Dans la mesure où les commutations entre processus légers, nécessaires à la réalisation du recouvrement, prennent du temps, le recouvrement ne peut pas être maximal, alors qu'il l'est pour une grande charge de calcul, cas où il n'y a qu'une commutation à chaque recouvrement. Enfin, le recouvrement pour les grands messages est moins bon que pour les petits. C'est dû au fait qu'il faut accroître la charge de calcul au fur et à mesure de l'accroissement de la taille des messages, pour toujours recouvrir la communication.

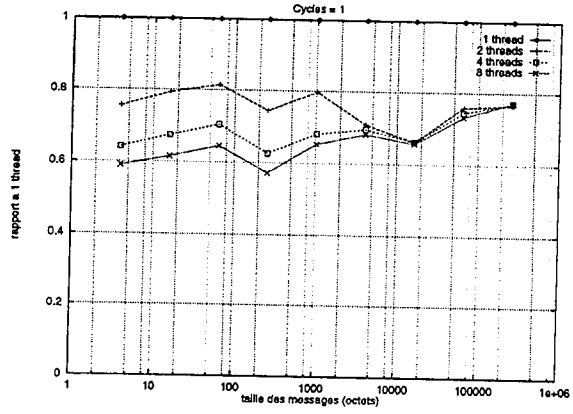

FIG. 4.34 - *Etude du recouvrement calcul-communication.*

Comparaison en fonction du nombre de processus légers et de la taille des messages (pas de charge de calcul)

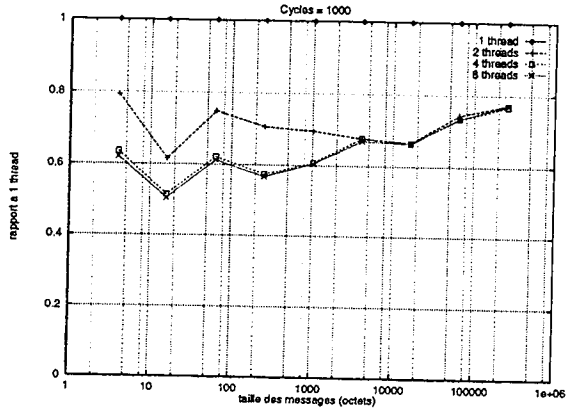

FIG. 4.35 - *Etude du recouvrement calcul-communication.*

Comparaison en fonction du nombre de processus légers et de la taille des messages (charge de calcul moyenne)

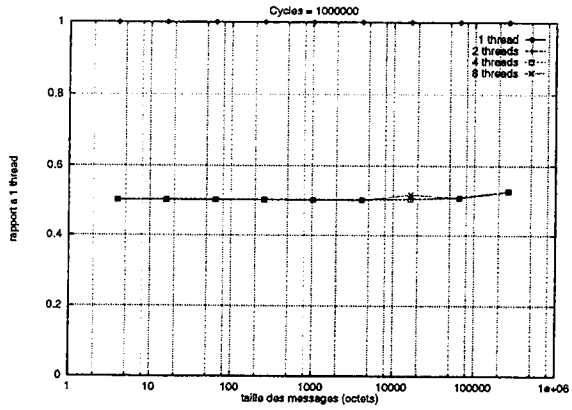

FIG. 4.36 - *Etude du recouvrement calcul-communication.*

Comparaison en fonction du nombre de processus légers et de la taille des messages (grande charge de calcul)

En fixant le nombre de processus légers (ici à 4), le graphique (fig. 4.37) confirme bien notre intuition : pour avoir un recouvrement optimal, il faut que la charge de calcul soit en proportion de la taille des messages. En partant du haut du graphique, les trois premières courbes présentent une charge de calcul trop faible : en conséquence, le recouvrement n'est pas complet, même pour les petits messages. Dans la quatrième courbe, la charge de calcul ($40\mu s$) représente une partie significative du coût de transmission du message : le recouvrement s'améliore. Dans les trois dernières courbes, le recouvrement est presque optimal, la charge de calcul étant suffisante, sauf pour les très grands messages.

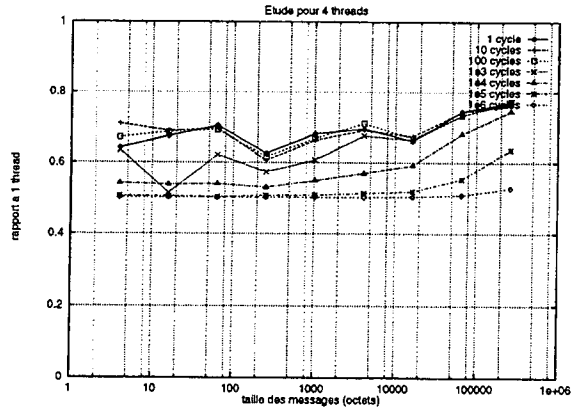

FIG. 4.37 - *Etude du recouvrement calcul-communication.*

Comparaison avec diverses tailles de messages et de calcul, pour 4 processus légers.

En conclusion de cette expérience, nous avons montré qu'Athapascan-0a est capable de recouvrir les attentes de résultats par du calcul utile, avec un petit nombre de processus légers. Le meilleur recouvrement est obtenu quand la granularité de calcul dépasse celle de communication. Si la granularité de calcul est très importante, il suffit d'avoir deux processus légers : en utilisant des communications asynchrones, il serait possible de recouvrir calcul et communication sans avoir à introduire le concept de processus légers, juste en lançant les communication « à l'avance » (voir à ce propos [Desprez & Tourancheau 1994]). Si la granularité de calcul est faible, il faut introduire un nombre conséquent de processus légers pour obtenir un recouvrement efficace. Cependant, le recouvrement ne

sera alors pas optimal, dû au nombre important de commutations nécessaires pour le réaliser. Notons qu'un nombre relativement limité de processus légers permet d'atteindre un bon recouvrement.

4.7.6 Comparaison de performances entre PVM et Athapascan-0a : une application réelle

Nous avons montré précédemment qu'utiliser Athapascan-0a entraînait un surcoût vis à vis de PVM, au niveau des mécanismes de base. Ce surcoût est le prix à payer pour disposer de la décomposition procédurale parallèle. Somme toute, il n'est pas excessif. On peut se demander ce qu'il en est au niveau d'une application réelle. Le problème d'une application réelle est que sa performance peut être fortement conditionnée par la politique et les mécanismes de régulation de charge. Malheureusement, le projet APACHE ne propose pas encore de mécanisme de régulation de charge. Cependant, nous pouvons donner un exemple de résultat, sans régulation de charge, établi sur une application réelle.

4.7.6.1 Multiplication de matrices formelles avec PAC++

Dans cet exemple, nous allons comparer une application multipliant des matrices formelles. Cette application est développée à l'aide de la bibliothèque PAC++, soit sur Athapascan-0a, soit sur PVM. Notons que la version PVM de la bibliothèque PAC++ est restreinte, PVM ne permettant pas d'entrelacer différentes activités. La bibliothèque PAC++ et l'exemple que nous citons ici ont été réalisés dans leur plus grande part par T.Gautier au sein du projet APACHE.

Dans le cas de la multiplication de deux matrices formelles de taille n x n, qui ont comme cœfficients des polynômes sur les entiers, de taille l bits, d'une seule variable et de degré maximum d, le coût arithmétique de l'algorithme est en $O(n^3 \times d^2 \times l^2)$; le coût des communication est en $O(n^2 \times d \times l)$. Le rapport est donc en $O(n \times d \times l)$, et peut donc être arbitrairement grand. Le but de cet exemple étant seulement de comparer Athapascan-0a avec PVM, nous avons mesuré le temps passé dans un produit de matrice (la méthode de parallélisation est le découpage en colonnes et circulation des lignes), sans faire intervenir aucune régulation de charge. La version Athapascan-0a et la version PVM de l'algorithme sont effectuées sur le même jeu de matrices.

taille, n	degré, d	bits, l	# proc	Athapascan sec.	PVM sec.	Surcoût %
10	5	64	8	1.74	1.34	30
10	10	64	8	4.24	4.16	2
20	5	64	8	11.7	10.4	13
10	5	64	2	3.5	3.21	9
20	5	64	4	17.4	16.3	7
20	10	64	8	29.9	29.5	1
40	5	64	8	52.7	49.1	7
10	10	64	2	13.8	14.6	-5
20	10	64	4	49.1	48.1	2
40	5	64	4	92.1	95.4	-3
40	10	64	8	147.9	148.7	-1
40	10	64	4	286.2	293.1	-2

TAB. 4.7 - *Comparaison d'Athapascan-0a et de PVM sur une application PAC++.*

Nous avons trié le tableau de résultats en fonction du rapport calcul / communication (petit rapport en haut). Les résultats montrent un surcoût très faible, dès que ce rapport est suffisamment grand. Les valeurs légèrement négatives (Athapascan-0a plus rapide que PVM) sont probablement dues à des différences, entre les deux versions, dans le tamponnage des messages.

Dans ce style de calcul intensif, très irrégulier, le facteur clé est l'adéquation de la régulation de charge et donc la possibilité pour le programmeur d'informer l'étage de régulation de charge des parallélisations possibles. Cela peut être aisément exprimé en PAC++ sur Athapascan-0a - chaque activité parallèle étant décrite par une procédure parallèle - et beaucoup plus difficilement avec PAC++ sur PVM, puisque les activités doivent être séquentielles, à moins de définir un automate chargé d'entrelacer au mieux les calculs. Cela est très complexe et équivalent à la réalisation d'un noyau de gestion de fils d'exécution. A l'opposé, l'efficacité effective des communications n'est pas très importante puisque le coût des communications peut être réduit à un petit pourcentage du coût total.

4.7.7 Exemples d'applications réelles

Outre l'évaluation d'un exemple de calcul formel donnée ci-dessus, nous présentons deux exemples d'applications réelles établies au-dessus d'Athapascan-0a.

4.7.7.1 Un système de programmation logique parallèle

Le système PloSys est un système de programmation logique, tirant parti du parallélisme implicite des applications Prolog et régulant la charge automatiquement. Le travail que nous survolons ici est celui de E.Morel [Morel 1996, Morel et al. 1996, Kannat et al. 1994], effectué dans le cadre du projet APACHE. Le système développé offre un Prolog complet et standard, indépendant de la machine cible, exploitant automatiquement le parallélisme « OU ». Cette forme de parallélisme consiste à traiter les deux branches d'une clause OU en parallèle. Pour cela, la pile d'exécution qui contient les points de choix et les liaisons entre atomes et valeurs est reproduite sur un second site, qui calcule alors la seconde branche du OU. C'est toujours le plus ancien point de choix qui est soumis à parallélisation. Les effets de bord sont pris en compte dans PloSys. En particulier, il y a exclusion lors du transfert de pile pour garder la cohérence des données.

Un mécanisme de régulation de charge très simple est réalisé dans un premier temps - le but du projet PloSys est d'étudier différentes politiques de régulation de charge. L'implantation choisie consiste en un ordonnanceur centralisé et un ensemble de travailleurs. Le transfert de charge est réalisé à l'aide d'Athapascan-0a : une demande de travail est émise par un travailleur vers l'ordonnanceur ; celui-ci répond par une indication de site surchargé. Une demande d'exportation de travail est ensuite effectuée par le travailleur libre à destination du site surchargé. L'exportation de charge en résulte généralement - mais il est possible que la demande d'exportation échoue si la charge du cite cible a changée entre temps. Une indication de leur nouvelle charge est enfin effectuée par les deux travailleurs à destination de l'ordonnanceur.

L'ordonnanceur aussi bien que les travailleurs sont découpés en un ensemble de fonctionnalités, chacune prise en charge par un processus léger différent. L'ordonnanceur offre une fonction de régulation de charge et une fonction de traitement des effets de bord. Il exporte à destination de l'utilisateur et des travailleurs les services suivants :

- initialisation et terminaison de la tâche,
- lancement de l'exécution,
- collecte de la charge des travailleurs,
- traitement des demandes de travail,
- collecte des traces d'exécution de l'ordonnanceur.

Les travailleurs sont constitués pour leur part de fonctions d'importation et d'exportation, de régulation de charge et d'évaluation Prolog. Ils offrent les services suivants :

- initialisation et terminaison de la tâche,
- évaluation Prolog,
- exportation de charge,
- collecte de traces,
- gestion de l'ordre séquentiel.

L'auteur montre sur certains exemple de bonnes efficacités, ainsi qu'une assez bonne régulation de charge. Nous voyons donc que le modèle d'exécution proposé par Athapascan- 0a convient pour un système de programmation logique parallèle :

- l'existence de processus légers permet de décomposer fonctionnellement le système de programmation, permettant ainsi de mieux isoler les différents composants,
- le modèle client-serveur offert par Athapascan-0a convient parfaitement pour la conception d'un protocole de régulation de charge et d'observation de l'exécution,
- la grande portabilité de l'exécutif parallèle facilite le portage de PloSys sur différentes machines.

Notons toutefois que l'absence d'interruption lors de la réception d'un message limite la prise en compte des demandes de travail. Actuellement, la scrutation du réseau est effectuée lors de l'envoi de l'état de charge. Cet envoi intervient lorsque la charge évolue de façon importante. L'indicateur de charge est le nombre de points de choix présents dans la pile d'exécution. Le test de la charge locale vis à vis d'un seuil d'émission peut être fait à chaque traversée d'un point de choix. Cependant, l'auteur indique qu'une solution moins coûteuse est un test périodique. Athapascan-0a n'est toutefois pas sauf vis à vis des signaux. Une communication par mémoire partagée est donc effectuée entre le traitant d'interruption périodique et le moteur d'évaluation Prolog. Un dernier problème est introduit par le mécanisme de coupure de Prolog. Pour une exécution efficace de ce mécanisme, il faudrait pouvoir interrompre le moteur Prolog dès l'arrivée d'un message de coupure. Cela n'est pas non plus possible puisqu'il n'y a pas d'interruption sur réception. Une solution, comme pour la prise en compte des demandes de travail, est de ne tester cet événement qu'à certains moments.

En conclusion, nous voyons qu'Athapascan-0a convient bien à PloSys. L'absence de temps partagé en particulier n'est pas un frein à une exécution efficace, de même que l'absence de priorités. Rendre Athapascan-0a *signal-safe* serait cependant peu coûteux et intéressant pour simplifier la synchronisation entre traitant d'interruption périodique et moteur Prolog. D'autre part, un noyau de communication, autre que PVM, remontant une interruption lors de l'arrivée d'un message, permettrait probablement une meilleure prise en compte des événements asynchrones, comme les demandes de travail et les indications de coupure.

4.7.7.2 Une application de dynamique moléculaire

Cette application, développée dans le cadre du projet APACHE par P.- E.Bernard [Bernard et al. 1996*b*, Bernard & Trystram 1996, Bernard et al. 1996*a*], calcule le comportement dynamique de molécules. Cette information aide les chercheurs pour connaître les propriétés des liquides, des solides, des gaz, ainsi que des protéines par exemple. Le principe de cette application est de calculer les positions des atomes en chaque instant en intégrant les équations du mouvement de Newton. On dispose donc des positions et des vitesses initiales des atomes, et l'on calcule les forces entre ceux-ci : les forces intramoléculaires qui lient les atomes à l'intérieur d'une molécule, et les forces d'interaction non liées, qui interviennent entre tous les atomes. Ces dernières forces sont les plus importantes en terme de charge de calcul. Pour éviter de devoir calculer les forces non liées entre tous les atomes, on effectue une approximation : les forces non liées sont supposées nulles si les atomes sont trop distants. On établi donc un *rayon de coupure*.

La charge de calcul doit être équilibrée entre les processeurs. Pour cela, une méthode de décomposition spatiale est appliquée : l'espace est découpé en cubes de même dimension que le rayon de coupure, et on associe des groupes de cubes à chaque processeur. L'association peut être aléatoire, cyclique sur une charge de calcul (le nombre d'atome) décroissante, ou récursif en gardant le meilleur équilibre de charge à chaque décomposition.

Le principe de calcul est donc le suivant. Pour chaque atome, on calcule en parallèle les forces non-liées dues aux atomes situés sur d'autres processeurs, et les forces intramoléculaires et les forces non-liées dues aux atomes situés sur le même processeur. Les équations du mouvement sont ensuite intégrées. Chaque processeur effectue donc une itération principale pour calculer la position des atomes qu'il gère, et exporte dans le même temps un point d'entrée permettant de calculer les forces exercées par les atomes qu'il gère, sur les atomes distants. Tous les processeurs disposants de cubes adjacents vont appeler ce point d'entrée pour calculer les forces induites par les atomes présents sur ce processeur. Symétriquement, ce processeur va appeler les points d'entrée offerts par les processeurs gérants des cubes adjacents pour établir les forces qu'ils appliquent sur ses atomes. On a donc un appel croisé, illustration immédiate de l'appel de procédure à distance asynchrone. Des synchronisations sont maintenues localement entre les fils d'exécution, pour garantir la cohérence des données : l'intégration du mouvement ne peut être faite que lorsque toutes les forces ont été calculées.

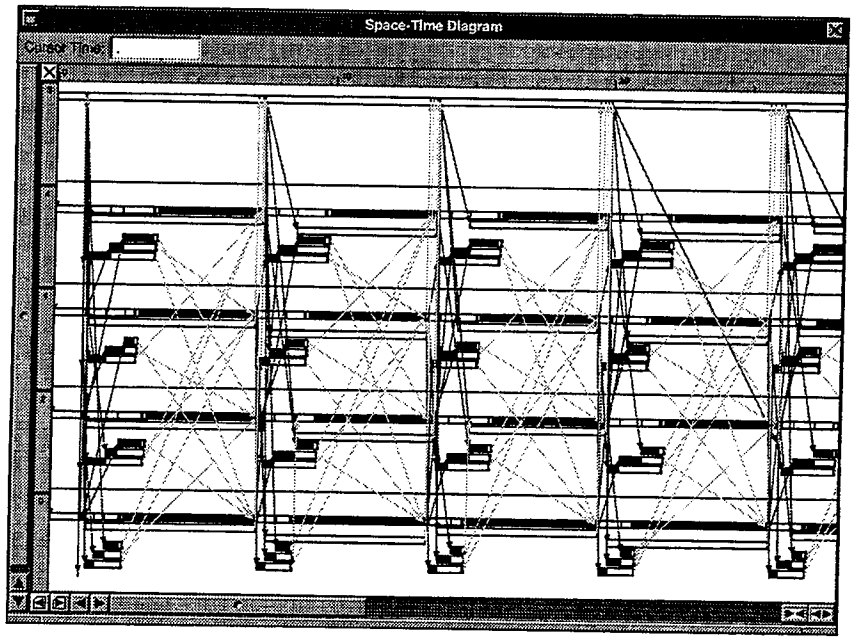

FIG. 4.38 - *Trace d'exécution.*

La trace d'exécution d'une application de dynamique moléculaire, réalisée au-dessus d'Athapascan-0a, est visualisée par un outil spécialisé. Elle montre les appels de procédures à distance et l'exécution des processus légers.

Nous donnons une trace d'exécution de cette application, établie avec l'outil de visualisation de traces développé dans le cadre d'APACHE. Les fils d'exécution sont les lignes horizontales (groupés par processeurs), les flèches en diagonales indiquent les messages de requêtes et de réponses. Les niveaux de gris indiquent l'état des processus : actifs, en attente. L'accélération atteinte par cette application est d'environ 10 pour 16 processeurs. Cette accélération dépend du placement effectué. Des expériences sont donc tentées sur le placement des cubes sur les processeurs, de façon à obtenir le meilleur équilibrage de charge.

Cette application est intéressante ; nous voyons que le RPC asynchrone fourni par l'exécutif Athapascan-0a est tout à fait adéquat. Il permet une expression simple de l'application, en évitant le recours à des mécanismes de communication asynchrone, ou d'une façon plus générale, à un automate. D'autre part, le partage de temps entre fils d'exécution est manifestement inutile pour cette application : les synchronisations liées au déroulement de l'application et au maintient de cohérence des données sont suffisantes pour établir l'ordonnancement de l'exécution. Enfin, le modèle de communication est double : les calculs faisant intervenir différents processeurs sont réalisés à l'aide de l'appel de procédure à distance. Par contre, à l'intérieur d'un processeur, les calculs sont réalisés à l'aide de la mémoire partagée entre tous les fils d'exécutions. Ce schéma d'organisation permet d'assurer la meilleure efficacité à l'application.

4.8 Conclusion sur le prototype

Le modèle de programmation d'Athapascan-0a, basé sur un mécanisme d'appel de procédure à distance asynchrone, générant des processus légers, permet une expression simple de la décomposition procédurale parallèle, indépendante de la régulation de charge employée, et autorise un recouvrement automatique des attentes de communication par du calcul. Le projet APACHE a démontré la validité de ce modèle pour les applications parallèles irrégulières.

L'interface d'Athapascan-0a présente un certain nombre d'opérateurs : définition, exportation, appel synchrone ou asynchrone d'un service, envoi et réception de données en tant que paramètres, gestion des machines physique et virtuelle et information, gestion mémoire, synchronisation locale. Comme tout prototype, l'interface d'Athapascan-0a présente un certain nombre de défauts de conception. Nous allons les énumérer brièvement.

- Le premier défaut, et peut être le plus important, réside dans le fait que nous n'avons pas assez séparé les *mécanismes* de l'*interface* de programmation. L'approche initiale a été de définir une sorte de langage de programmation, dont les constructions MACROS sont un résidu. Nous n'avons jamais eu l'ambition de construire un compilateur, aussi cette notion de langage est-elle superflue. En particulier, les macros posent problème lors de la compilation, par exemple en C++, et lors du déverminage. Avec le recul, nous pensons qu'une bien meilleure approche pour ce genre d'exécutif est celle d'une bibliothèque de fonctionnalités.
- Un autre défaut qui nous semble maintenant apparaître est le fait que nous avons complètement caché PVM. Dans un objectif de projet spécifique tel qu'APACHE, cela est légitime, mais Athapascan-0a aurait sûrement pu bénéficier d'une meilleure diffusion à l'extérieur du projet s'il s'était présenté comme une sur-couche de PVM, n'empêchant pas les applications usuelles de PVM de fonctionner et ne nécessitant pas un apprentissage spécifique. Nous avons évalué pendant quelques temps cette approche comme successeur du prototype, pour finalement choisir MPI comme nouveau noyau de communication. Mais cela est une autre histoire.
- Un ensemble d'erreurs de conception ou de réalisation peuvent être citées : par exemple, **LoadPVMTask** devrait avoir une variante asynchrone ; **DoCall / DoSpawn** et **LoadPVMTask** devraient avoir une variante multiple pour bénéficier d'une optimisation d'exécution multiple (réalisation sur un arbre couvrant) ; **LoadPVMTask** devrait admettre des arguments de commande etc... Tous ces défauts sont mineurs et pourraient facilement être corrigés ; cependant le destin d'un prototype est d'être imparfait...
- De façon plus générale, un ensemble de fonctionnalités nous semble être utile comme extension du prototype : nous pouvons citer par exemple un mécanisme de garde des points d'entrée pouvant prendre en compte les arguments des requêtes, un mécanisme de synchronisation procédurale, d'autres mécanismes de synchronisation sur la mémoire locale (par exemple *multiple readers, one writer*), une définition de zones mémoire accessibles à distance ... Certains de ces points sont repris et présentés plus en détail dans le chapitre suivant.

Un autre défaut réside dans le fait, nous semble-t'il, qu'Athapascan-0a ne repose pas sur une couche stricto-sensu « portable ». Athapascan-0a est fortement relié à PVM ; nous n'avons pas pu (mais ce n'était pas l'un de nos objectifs de recherche) le porter tel quel sur MPI. Nous pensons qu'un tel exécutif parallèle devrait reposer sur une interface de portabilité clairement définie, générale et efficiente. Le portage sur une nouvelle machine consisterait alors à porter cette interface et non l'exécutif lui-même. Athapascan-0a n'est tel quel portable que sur un système présentant l'interface de PVM, or cette interface n'est pas idéale pour atteindre de bonnes efficacités sur certaines machines parallèles. MPI nous semble avoir balayé la mode PVM ; peut être la mode MPI sera-t'elle balayée à son tour ; en tout cas la définition d'une interface de portabilité générale et efficiente nous semble le préalable à la construction d'un exécutif réellement portable.

Athapascan-0a repose sur un modèle d'interaction à base de RPC. Toutefois ce modèle d'interaction et celui par échange de messages ne nous semblent pas contradictoires, et même plutôt complémentaires. Le mécanisme client-serveur permet de prendre en compte l'irrégularité et l'échange de messages permet de profiter des aspects réguliers des applications. Il est facile d'étendre Athapascan-0a pour que ses fils d'exécution puissent communiquer par échange de messages, par exemple en utilisant des étiquettes différentes comme désignation de portes de communication pour chaque fil.

Nous pensons maintenant que tout exécutif parallèle pour applications irrégulières doit fournir cette double fonctionnalité de communication.

Enfin, il est évident qu'il reste encore beaucoup de travail pour définir une interface applicative adéquate pour la programmation d'applications parallèles irrégulières, en particulier si elles doivent être efficacement portables. C'est là le thème de recherche de la couche Athapascan-1 ; ce n'est donc pas réellement le problème ici sauf qu'une telle interface influe sur l'ensemble des fonctionnalités que doit offrir l'exécutif sous-jacent. Tant qu'une telle interface n'aura pas été validée, nous ne serons pas certains des fonctionnalités à mettre dans l'exécutif de base.

L'implantation du support exécutif Athapascan-0a, au dessus d'un noyau de communication de même API que PVM et d'un noyau de multiprogrammation non préemptible, fut simple. Le code source de PVM n'a pas été modifié ; PVM et différents noyaux de processus légers furent seulement mis côte à côte. Cela nous a permis d'utiliser PVMe, dont le source n'est pas public. Comme les processus légers ne sont pas préemptibles et que leur ordonnancement est sous contrôle, le processus léger courant utilise toujours les fonctions PVM en exclusion mutuelle. La facilité fournie par PVM 3 pour employer des tampons multiples fait le reste du travail. Les messages de requête sont envoyés directement par le processus léger qui appelle. La réception des messages de réponses, aussi bien que la réception des messages de requête est faite comme décrit précédemment, lors d'opérations d'ordonnancement-
scrutation des processus légers.

Ce prototype a été porté sur un nombre conséquent de machines et systèmes. Malgré la diversité de comportement des noyaux sous-jacent, le surcoût introduit par Athapascan-0a, au niveau des caractéristiques de base, est faible. L'efficacité du recouvrement *automatique* des attentes de communication par du calcul a été démontré. L'expérimentation sur une application réelle confirme le faible surcoût. L'utilisation d'une couche de régulation de charge, rendue possible par le modèle spécifique d'Athapascan-0a, reste cependant à évaluer.

Ce prototype a satisfait ses objectifs initiaux : il nous a permis de juger l'approche initiale du projet APACHE et de démarrer les autres travaux ; nous avons montré qu'il permet d'exprimer des applications parallèles irrégulières ou des supports exécutifs de haut niveau ; qu'il est facilement extensible, observable et maniable.

En particulier, les deux objectifs principaux d'un tel exécutif parallèle pour applications irrégulières, la portabilité et l'efficacité, sont assurés. La portabilité est assurée par la demande très faible sur les fonctionnalités d'un noyau de multiprogrammation non préemptible et la présence d'un noyau de communication présentant l'API de PVM. Il est très facilement portable, quasiment sans modification du source, puisque PVM est dors et déjà largement porté et que l'adaptation à un noyau de multiprogrammation un tant soit peu conventionnel ne prend généralement qu'une journée. Il est efficace, comme nous l'avons vu, vis à vis de PVM domaine public. Il est aussi efficace puisqu'il autorise l'emploi de versions optimisées ou réécrites de PVM, pour des machines spécifiques, là encore sans perte excessive. Enfin, il permet le développement d'applications efficacement portables, par l'emploi des concepts de poly-algorithmes et de décomposition procédurale parallèle.

Ce support exécutif permet la réalisation d'applications parallèles irrégulières ; il est la base de travail du projet APACHE. Il supporte en particulier la bibliothèque de calcul formel PAC++, l'ébauche de compilateur PROLOG, l'application de dynamique moléculaire et divers autres composants du projet APACHE comme la réexécution déterministe, la prise de trace et la visualisation, la régulation de charge. En tant que substrat du projet APACHE, il est documenté et son évolution est suivie et mise à disposition de tous les membres du projet. Une version d'Athapascan-0, en cours de réalisation sur MPI, a grandement profité de cette réalisation.

Au delà de l'intérêt de ce support d'exécution pour le développement du projet APACHE, les

apports théoriques de ce travail sont multiples :

- la démonstration, par une réalisation pratique, de la validité de l'approche de la multiprogrammation légère accolée à un noyau de communication portable comme PVM ou MPI, dans le cadre spécifique des applications parallèles irrégulières. En particulier nous avons montré qu'une telle réalisation est facilement possible et efficace vis à vis des communications natives,
- l'identification du problème lié au couplage multiprogrammation légère / communications en l'absence de réception de message par interruption : il faut inventer un algorithme d'ordonnancement-
  scrutation adéquat, c'est à dire qui n'induise pas un surcoût trop important et qui permette une progression efficace, aussi bien des communications que des calculs,
- le dernier apport de ce travail est d'avoir permis une illustration effective d'applications parallèles décrites suivant le concept assez original de poly-algorithmes et de décomposition procédurale parallèle. En particulier, nous mettons en avant un modèle de calcul parallèle dans lequel la concurrence est maîtrisée par le contrôle des commutations (pas de préemption « sauvage »). Cette illustration sera approfondie par les autres axes de recherche du projet APACHE.

Chapitre 5

Comparaison d'Athapascan-0 à d'autres exécutifs parallèles

Nous avons montré la nécessité d'un support exécutif pour applications parallèles *irrégulières*. Nous avons, par notre travail dans le projet APACHE, développé un tel exécutif à la fois portable, souple et efficace. Nous allons maintenant présenter divers exécutifs qui peuvent être comparés à Athapascan-0, soit parce qu'ils ciblent eux aussi des applications parallèles irrégulières, soit parce qu'ils emploient les mêmes techniques (communications par échange de messages et multiprogrammation légère). Notons que la quasi-totalité de ces exécutifs ont été développés *postérieurement* à la première implantation d'Athapascan-0a[1] . Ce fait est en soi un gage de l'intérêt de notre propre recherche. Comme nous allons le voir, beaucoup de ces exécutifs sont basés sur PVM et utilisent une organisation semblable à celle d'Athapascan-0a. La présentation de ces exécutifs suit une démarche analytique : chacun est détaillé dans le même ordre, celui adopté pour la présentation d'Athapascan-0a ; les différentes options de conception peuvent ainsi être comparées. Nous terminons cette présentation par une comparaison assez exhaustive, sur ce que nous considérons être les caractéristiques les plus importantes, puis par une conclusion générale sur ce type de support exécutif.

Les supports d'exécution que nous présentons ici sont les suivants :

TPVM, PM^2, DTh, DTS, LPVM, DTMS, MPI-F, Nexus, Chant, Panda.

Nous avons aussi étudié la proposition du groupe de travail Ports, la proposition d'extension de MPI (MPI-2) et le portage d'une implantation de MPI sur Nexus (MPI-CH/Nexus), qui sont tous reliés à l'intégration de la multiprogrammation légère et des communications. Cette comparaison n'est pas exhaustive de tous les supports d'exécution existants à l'heure actuelle ; nous pouvons aussi citer différents travaux concernant PVM et les processus légers : Pt-PVM [Krone et al. 1995], LPWP [Chuang 1994], PVMt [Ackaouy 1994],NewThreads [Felton & McNamee 1992*b*, Felton & McNamee 1992*a*], ARCH [Adamo 1996], que nous ne présentons pas ici.

5.1 TPVM

TPVM[2] [Ferrari & Sunderam 1994, Ferrari & Sunderam 1995*a*, Ferrari & Sunderam 1995*b*], développé par Adam Ferrari (Université de Virginie) et Vaidy Sunderam (Université d'Emory), est une sur-couche de PVM autorisant l'utilisation de processus légers avec PVM. Les buts de TPVM sont de définir les processus légers comme unité de parallélisme, à la fois pour obtenir un parallélisme de grain plus fin, une décomposition plus naturelle des problèmes, mais aussi pour recouvrir les calculs et

1. Une exception notable est Panda, qui est antérieur.
2. http ://uvacs.cs.virginia.edu/~ajf2j/tpvm.html

les communications, faciliter la programmation de calculs concurrents asynchrones, éventuellement auto-ordonnancés et enfin, permettre une utilisation plus efficace des multiprocesseurs symétriques.

La bibliothèque PVM n'est pas modifiée, mais complétée par des mécanismes qui permettent un parallélisme de grain plus fin, éventuellement adaptatif, qui facilite la régulation de charge ; ce parallélisme est soit du type processus communicants (les communications entre processus de PVM sont étendues à des communications entre processus légers dans TPVM), soit du type parallélisme événementiel (les auteurs l'appellent *macro-dataflow*). Un mécanisme d'accès mémoire distant est aussi fourni. TPVM se décompose en quatre modules : la bibliothèque d'extension d'interface de PVM, un exécutif gérant la multiprogrammation, une interface de processus légers portable et un processus *threads server* qui gère l'ordonnancement et les données globales.

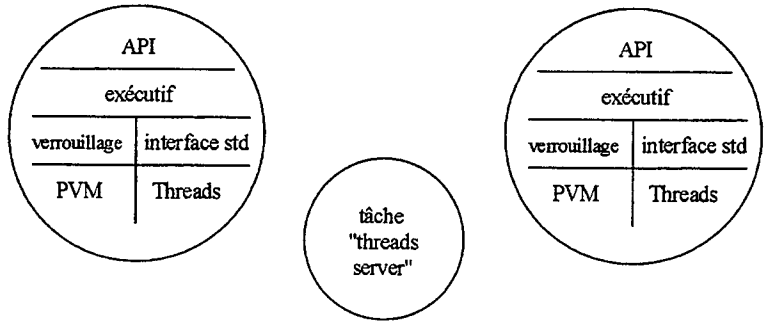

FIG. 5.1 - *TPVM : les composants.*

Bien que TPVM propose des processus légers dans une même espace d'adressage, les auteurs ne considèrent pas la possibilité de communication ou synchronisation par mémoire partagée entre processus légers. Les tâches de PVM sont considérées comme des réceptacles pour les processus légers ; on les instancie comme dans PVM. Cependant les communications se font uniquement entre processus légers. Chaque processus léger est identifié globalement de façon unique. Ces identificateurs peuvent être transmis entre processus légers. L'identificateur contient l'adresse du site où réside le processus et permet donc sa localisation. TPVM n'offre pas la migration de processus légers.

Les fonctions de communication sont préfixées par *tpvm_* et sont semblables à celles de PVM. Par exemple *tpvm_initsend*, *tpvm_pkXXX*, *tpvm_send*, *tpvm_mcast*, tpvm_*recv*, *tpvm_nrecv*, *tpvm_upkXXX*, etc... existent et effectuent les mêmes actions que dans PVM sauf qu'ici les communications se font entre processus légers. La création d'un processus léger se fait en deux étapes : l'exportation d'un point d'entrée et l'instanciation de ce point d'entrée. L'exportation se fait avec la primitive *tpvm_export*, qui précise le nom symbolique du point d'entrée, la fonction qui sera exécutée et le modèle dans lequel s'inscrit ce point d'entrée : processus communicants ou bien flot de données. Dans le modèle flot de données, on doit en plus préciser un ensemble d'étiquettes qui régiront l'activation du point d'entrée. La méthode d'instantiation d'un point d'entrée est différente selon le modèle. Dans le modèle processus communicants, la primitive *tpvm_spawn* simule *pvm_spawn* pour les processus légers. On précise donc le nom du point d'entrée appelé, la localisation des nouveaux processus légers (soit dans la même tâche que l'appelant, soit dans une tâche particulière, soit enfin au choix du système) et le nombre de processus légers à créer. Cette primitive, comme dans PVM, retourne les identifications des processus créés. L'instanciation d'un point d'entrée est donc réellement la création d'un processus léger et non un appel de procédure à distance. Il n'y a pas de paramètres transmis lors de la création ; mais il est possible d'en transmettre par la suite par des communications « normales » entre les processus légers.

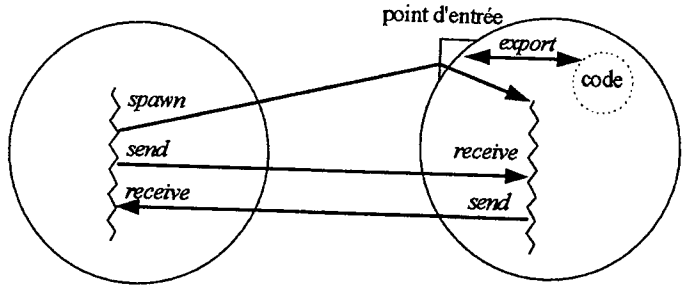

FIG. 5.2 - *TPVM : le modèle processus communicants.*

Dans le modèle flot de données, on doit construire un message. Ce message est envoyé non pas par la primitive *tpvm_send*, mais par *tpvm_invoke*, qui ne précise pas le site d'exécution, mais un certain nombre d'étiquettes identiques. Une correspondance non déterministe est effectuée entre les étiquettes précisées lors de l'*export* et celles de chaque *invoke*. Quand les règles de l'exportation sont vérifiées (ie. chaque étiquette est satisfaite par un message), un processus léger correspondant est créé, sur un site choisi par le système. Ce processus léger peut recevoir (par *tpvm_recv*) tous les messages qui ont contribués à sa création.

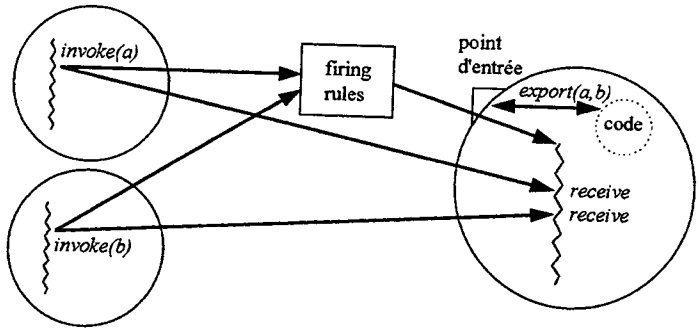

FIG. 5.3 - *TPVM : le modèle data-flow.*

Des services particuliers d'écriture et de lecture de mémoire à distance sont fournis. La zone mémoire concernée doit être exportée (*tmp_exportmem*) ; pour cela elle est nommée par un nom symbolique et un numéro de fragment et typée par un type de base et une étendue. Les accès à un fragment de mémoire distante se font par un copie (*tpvm_rcpy*) dans un sens ou l'autre (lecture ou écriture). Notons que seul le fragment en entier peut être accédé et cela en exclusion mutuelle. De plus il est dense. D'autres services particuliers permettent l'écriture en exclusion sur la console et sur un fichier journal.

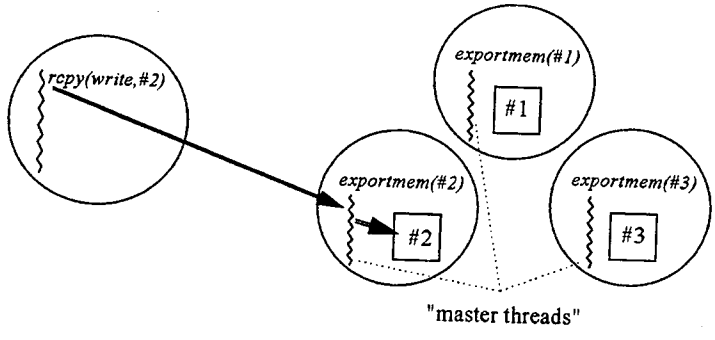

FIG. 5.4 - *TPVM : l'accès mémoire à distance.*

Il n'y a pas de modèle d'exécution particulier. Par exemple, aucun opérateur n'est prévu pour la terminaison d'un programme TPVM. Il est cependant possible de terminer le processus léger en cours (*tpvm_exit*), ou le *main()* d'un programme (*tpvm_go*), sans pour autant intervenir sur les autres processus légers. De façon similaire, il est possible de terminer une tâche et tous les processus légers qu'elle contient (*tpvm_killtid*). Toute exportation peut être annulée (*tpvm_unexport* et *tpvm_unexportmem*) ; cette opération est faite automatiquement lorsqu'on termine une tâche : toutes les exportations définies par cette tâche sont annulées. Enfin il est possible de synchroniser certains processus légers, à l'aide d'une barrière « légère » identifiée par une clé numérique (*tpvm_threadsync*). L'introduction de cette primitive évite le recours aux groupes de PVM.

TPVM a été construit à l'aide du noyau de processus légers REX. Il a ensuite été porté sur Solaris, et plus récemment sur les C-threads pour l'exécution sur Aix. L'interface de processus légers portable ne défini que les opérateurs suivants : création, suicide et identification de processus légers, l'action de relâchement du processeur (*yield*) et l'exclusion mutuelle. Sur Solaris la préemption est activée, il est donc nécessaire d'assurer l'exclusion mutuelle sur les fonctions qui ne sont pas réentrantes. En particulier, toutes les fonctions de la bibliothèque TPVM sont exécutées en exclusion mutuelle (par un verrou global). L'exclusion n'est pas assurée sur un noyau non préemptif, puisque de toute façon le processus léger en cours est forcément en exclusion.

Au niveau implantation, un processus léger particulier, dit *thread server*, centralise les définitions d'exportation de points d'entrée et de mémoire, gère les étiquettes postées pour le mécanisme d'instantiation flot de données et gère le choix d'un site pour les instanciations qui ne sont pas directement placées. Actuellement, le choix d'un site est effectué en tourniquet sur tous les sites. La réalisation est centralisée, mais pourrait évoluer vers un réalisation distribuée ultérieurement. A côté de ce processus léger particulier, unique, chaque tâche possède un processus léger dit « maître » qui effectue les communications unilatérales : principalement les créations de processus léger dans la tâche et les requêtes de lecture et d'écriture pour une zone mémoire de la tâche. Par exemple, l'activation d'un point d'entrée sur un site non déterminé, en mode flot de données, est effectué comme suit : les processus légers qui veulent effectuer cette activation transmettent à leur processus maître le message à envoyer ; celui-ci appelle le *thread server* qui mémorise les étiquettes fournies et, quand une règle d'activation est satisfaite, détermine le site d'activation (une tâche où le point d'entrée a été défini comme exporté), puis retransmet la requête au processus maître de la tâche cible. Le processus maître active le point d'entrée (crée le processus léger), en informe le *thread server* et lui indique l'identification du processus créé. Celui-ci peut alors répondre à tous les processus maîtres des processus activateurs l'identité du processus créé, de façon à ce que chacun puisse lui envoyer son message. Le processus activé peut alors recevoir chaque message tour à tour. La primitive *tpvm_invoke* est donc non bloquante, comme *tpvm_send*.

Le système TPVM fonctionne sur PVM standard. Le *tag* de chaque message PVM sert à identifier les processus léger émetteurs et destinataires. Pour cela, le tag PVM, de 32 bits, est découpé comme suit : 16 bits servent à encoder un tag pour les communications entre processus légers, 8 bits encodent le numéro local du processus léger destinataire du message (l'identité de la tâche destinatrice est stockée séparément par PVM) et enfin 8 bits encodent le numéro local du processus léger origine (de même l'identité de la tâche origine est stockée séparément par PVM). Chaque processus léger a une file de messages qui lui sont destinés mais non encore pris en compte. L'émission d'un message est faite directement, en exclusion, par le processus léger émetteur. Si le processus léger destinataire est dans la même tâche, le message est simplement mis dans sa file, sans devoir traverser PVM. La réception d'un message peut se faire comme pour PVM, en filtrant la source ou l'étiquette du message. Durant la réception d'un message (en exclusion), un processus léger peut recevoir un message destiné aux autres processus légers ; il est alors stocké dans la bonne file. L'algorithme de réception est donc schématiquement le suivant :

```
rechercher (tag, source) dans la file des messages pour le processus léger qui reçoit
si trouvé sortir
sinon
  répéter infiniment
    si un message peut être reçu de façon non-bloquante par pvm_nrecv
      si c'est le bon message sortir
      sinon le mettre dans la file du processus léger destinataire
    sinon
      si certains processus légers du site ne sont pas bloqués en attente de messages
        se considérer comme bloqué en attente de message
        relâcher le processeur (yield)
        rechercher (tag, source) dans la file des messages pour le processus léger
        si trouvé sortir
      sinon
        effectuer une réception bloquante pvm_recv
        si le message reçu est le bon sortir
        sinon, le mettre dans la file du processus léger destinataire
```

FIG. 5.5 - *TPVM : ordonnancement-scrutation.*

Donc, soit le message a déjà été reçu, soit il faut regarder sur le réseau. Si le réseau ne fournit rien mais que d'autre processus légers soient prêts à tourner, on relâche le processeur pour les laisser tourner. Si aucun autre processus léger n'est prêt à tourner, on effectue une réception bloquante, ce qui permet « d'arrêter » le processus lourd. Notons qu'il est nécessaire, après avoir relâché le processeur et avoir repris la main, de parcourir à nouveau la file des messages non pris en compte, puisque cette file a pu être modifiée par un autre processus léger. Lors du yield, il est nécessaire de libérer le verrou global pour éviter un interblocage, puis de le reprendre tout de suite après le yield pour se retrouver en exclusion.

TPVM est porté sur SunOS4, avec REX ; sur Solaris, avec les processus légers Solaris ; et sur Aix, avec les C-threads. Les expériences menées par les auteurs montrent que la création de processus légers est nettement plus rapide que celle de tâches PVM. Le surcoût de démultiplexage dans les communications est important (15% pour des messages d'1Ko), et n'est pas constant mais semble logarithmique (voir figure 5.6) ; d'autre part les communications entre deux processus légers dans la même tâche sont assez longues (malgré le fait de ne pas passer par PVM, c'est seulement un peu plus de deux fois plus rapide que les communications à distance, voir figure 5.7). Les auteurs rapportent une accélération de 10% par rapport à PVM sur une application qui ne peut être efficacement écrite sur le nombre de processeurs disponibles. Les auteurs indiquent aussi que TPVM peut être contre-performant sur des applications SPMD régulières. TPVM est un prototype d'évaluation, sur lequel quelques applications ont été réalisées. Une évolution possible est son intégration à PVM 3.4.

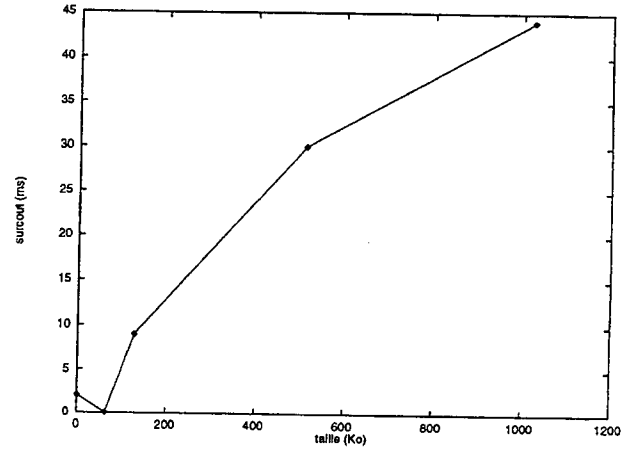

FIG. 5.6 - *TPVM : surcoût*

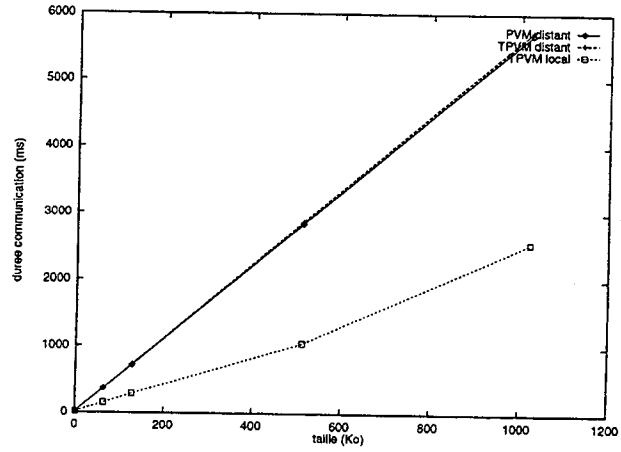

FIG. 5.7 - *TPVM : comparaison communication locale / distante.*
(courbes tirées de mesures publiées dans [Ferrari & Sunderam 1995a])

Comparé à Athapascan-0a, TPVM utilise un tout autre modèle de programmation (il n'y a ni RPC ni partage de mémoire entre les processus légers). TPVM peut utiliser le partage de temps ; nous avons vu qu'Athapascan-0a pourrait aussi l'utiliser. TPVM présente un surcoût important ; probablement à cause d'une consultation trop fréquente de la tâche *thread server*, centralisée. D'autre part la scrutation des messages est effectuée tour à tour par chaque processus léger ; si un message attendu ne se présente pas, le processus léger effectue un nombre important de *yields*. Enfin, la désignation des points d'entrée est symbolique, ce qui empêche toute liaison simple de modules de provenances diverses (risque de double définition de points d'entrée).

5.2 PM²

L'environnement de programmation parallèle PM² (*Parallel Multithreaded Machine*)[3] [Namyst & Méhaut 1995*c*, Namyst & Méhaut 1995*b*, R.Namyst & J-F.Méhaut 1995] du projet ESPACE (*Execution Support for Parallel Applications in high-performance Computing Environments*) de l'équipe GOAL (Groupe Objets et Acteurs de Lille) de l'Université de Lille répond à des buts semblables à

3. http ://gordon.lifl.fr/~namyst/pm2.html

ceux d'Athapascan : supporter des applications parallèles irrégulières - en particulier présentant un grand nombre de tâches de grain fin, permettre le recouvrement des attentes de communication par des calculs, réguler dynamiquement la charge, aider l'analyse de performances et la mise au point.

L'implantation de PM^2 est actuellement faite au-dessus d'un noyau de multiprogrammation maison, MARCEL et de PVM domaine public. Dans la mesure où MARCEL supporte la préemption, PVM a été modifié pour autoriser une exécution concurrente de plusieurs actions de communication. Nous reviendrons sur cette implantation dans la suite. MARCEL [Namyst & Méhaut 1995*a*] est un noyau de multiprogrammation de niveau utilisateur porté sur quelques architectures de machines (Sparc - SunOS4, Sparc - Solaris, Alpha - OSF/1, Intel - Linux et Power - Aix), qui présente une interface POSIX et des fonctionnalités spéciales, en particulier l'instrumentation du nombre de processus légers prêts, l'extension de pile et la migration de processus légers. Ces fonctionnalités sont utiles pour la programmation d'applications combinatoires : l'extension de pile autorise un support plus facile de la récursivité et l'instrumentation et la migration sont intéressantes pour certaines formes de régulation de charge. L'extension de pile se fait à travers quelques primitives : tester l'espace restant dans la pile, enregistrer / désenregistrer un pointeur sur une donnée stockée dans une pile, étendre la pile d'une certaine quantité d'octets. L'extension de pile consiste à réallouer la pile avec une taille supérieure, en changeant tous les pointeurs sur des données situées dans l'ancienne pile pour les faire pointer sur la nouvelle. Notons que ce mécanisme peut être coûteux en temps et n'est pas transparent pour le programmeur (il ne faut pas oublier d'enregistrer un pointeur concerné, ni copier un tel pointeur...) ; une extension de pile par allocation de nouveaux fragments serait peut être plus efficace et transparente, au prix d'une petite modification du compilateur. La migration est limitée à des architectures homogènes et des processus de structure homogène ; toutes les entités que doit voir un processus léger migrable doivent être répliquées sur tous les sites. Il est possible de déclarer des processus légers migrables et d'autre non, de s'enquérir des processus légers migrables et de migrer un ensemble de processus légers à la fois pour amortir les coûts d'initialisation de l'opération. L'opération de migration consiste principalement à envoyer la pile du processus léger sur le site à distance ; c'est une opération analogue à une extension de pile et la même philosophie est employée. Notons que la migration pose un problème lorsqu'elle intervient durant un RPC asynchrone, le processus léger à qui les résultats sont destinés pouvant avoir migré durant l'exécution du service. Les auteurs de PM^2 ne traitent pas ce cas, il est donc nécessaire de déclarer le processus léger non migrable lorsqu'il effectue un RPC asynchrone. MARCEL supporte le temps partagé, avec une notion de priorité de processus légers telle qu'une priorité de 3 représente trois fois plus de temps d'exécution qu'une priorité de 1 (si les processus légers sont dans le même processus).

Le modèle de programmation de PM^2 est assez proche de celui d'Athapascan : le mécanisme d'interaction de base est l'appel de procédure à distance dit « léger » (LPRC). Ce mécanisme se décline en plusieurs variantes : une variante synchrone qui correspond à un RPC classique, une variante sans retour qui correspond à un RSR, et une variante à attente différée qui est un RPC asynchrone. Les procédures appelables à distance définissent des points d'entrée ; elles sont déclarées avant le démarrage proprement dit du programme parallèle ; l'identification est faite par un numéro et tous les sites doivent déclarer les mêmes services dans le même ordre car l'association d'un numéro à un service est faite automatiquement. Les paramètres de ces procédures se présentent sous la forme de deux structures : une pour les paramètres d'entrée et une pour ceux de sortie. L'emballage et le déballage de ces structures vers le réseau est effectué automatiquement aussi bien du côté du client que de celui du serveur par l'écriture préalable de talons (4 au total, pour satisfaire toutes les combinaisons client/serveur x emballage/déballage). Les talons utilisent directement les primitives PVM pour copier les données dans les tampons de communication. PM^2 supporte donc l'hétérogénéité. Lors d'un LRPC, l'exécution de la procédure appelée est effectuée soit directement par le démon de communication (mode *quick*, il est alors nécessaire de ne pas effectuer d'actions bloquantes pour ne pas risquer de produire un interblocage), soit par un processus léger créé spécialement pour l'occasion (mode normal, on peut

alors spécifier sa priorité et la taille de sa pile à l'appel). PM² propose un mécanisme d'exceptions pour la reprise d'erreur ; ce mécanisme général est en particulier employé pour les erreurs pouvant survenir des appels LRPC. Trois types d'exceptions sont levées dans ce cas : CONSTRAINT_ERROR lorsque la priorité indiquée est invalide, STORAGE_ERROR lorsqu'il est impossible de créer un processus léger sur le site spécifié, LRPC_ERROR lorsqu'une exception survient durant l'exécution du service distant. Notons que PM² fait un usage important de macro-instructions ; son intégration dans d'autres langages que C peut donc poser les mêmes problèmes que pour Athapascan-0a. Notons aussi qu'il est possible de gérer une machine virtuelle que l'on peut agrandir ou restreindre à volonté.

Alors que le modèle de programmation est somme toute assez semblable à celui d'Athapascan-0a, l'implantation diffère beaucoup puisque PM² supporte l'exécution préemptive en temps partagé. Pour cela, le code source de PVM a du être modifié pour être rendu réentrant. Les modifications apportées concernent principalement le partage de l'accès au réseau et l'exclusion lors de l'allocation mémoire et de la gestion des tampons de communication. L'accès au réseau se fait en exclusion :

- tous les transferts se font en exclusion (exclusion de transfert)
- il y a aussi exclusion sur les données de contrôle (exclusion de contrôle)

Plusieurs fils d'exécution peuvent tenter une réception en même temps ; cependant un seulement entre dans la portion de code en exclusion. Les autres lui communiquent les données à envoyer par l'intermédiaire d'un *pipe* et se synchronisent à l'aide de sémaphores. L'unique fil d'exécution en mode transfert fait :

- soit l'émission de son message,
- soit l'attente de réception de son message, avec la réception des messages pour les autres et l'émission des messages des autres (qui lui sont transmis par l'intermédiaire du pipe), tant que l'attente n'est pas finie.

Plus précisément l'algorithme d'émission d'un message est le suivant :

```
début d'exclusion de contrôle
si réception en cours
  créer un sémaphore,
  créer une requête contenant l'identité du message et du sémaphore,
  envoyer cette requête sur le pipe,
  fin d'exclusion de contrôle
  attendre la fin d'émission sur le sémaphore
sinon
  début d'exclusion de transfert et fin d'exclusion de contrôle
  transférer (émettre) le message
  fin d'exclusion de transfert
```

FIG. 5.8 - *PM² : algorithme d'émission.*

L'algorithme de réception d'un message est le suivant :

```
début d'exclusion de contrôle
indiquer réception en cours
début d'exclusion de transfert et fin d'exclusion de contrôle
transférer (recevoir) le message,
puis épuiser les requêtes restant dans le pipe
indiquer réception terminée
fin d'exclusion de transfert
```

FIG. 5.9 - *PM² : algorithme de réception.*

L'algorithme de transfert d'un message entrant ou sortant est le suivant :

```
répéter
  répéter
    select sur le canal sortant ou les canaux entrants (dont le pipe)
    si aucun canal n'est actif : yield
  jusqu'à ce qu'un canal soit actif
  si un message est reçu
    s'il provient du pipe :
      préparer son émission sur le réseau
    sinon :
      le stocker normalement dans les files de messages
  si un message peut être émis
    l'émettre
    éventuellement, libérer le sémaphore associé
jusqu'à ce que le message soit émis ou reçu
```

FIG. 5.10 - *PM² : algorithme de transfert.*

En résumé, un fil d'exécution peut monopoliser la réception longtemps, mais :

- il reçoit tous les messages et les stocke dans les files, donc ne bloque généralement pas les autres fils d'exécution qui tentent des réceptions, car ces autres fils d'exécution trouvent les messages attendus dans les files et ne se bloquent donc (généralement) pas en réception ;
- il émet les messages pour le compte des autres fils d'exécution, qui lui parviennent par l'intermédiaire du *pipe*.

Vis à vis de TPVM, la réalisation de ce fonctionnement est plus compliquée (il y a modification de PVM), mais le résultat semble être comparable : durant une réception bloquante, d'autres émissions et réceptions peuvent progresser. L'efficacité comparée de chacune de ces deux solutions reste cependant à mesurer. Notons quand même que TPVM est construit de sorte que chaque fil d'exécution utilise un *tag* différent ; alors que dans PM², n'importe quel fil d'exécution peut utiliser n'importe quel *tag*. Dans TPVM, l'exclusion de transfert s'interrompt lorsqu'un *yield* est fait ; l'utilisation des opérateurs *pvm_recv* et *pvm_nrecv* de PVM est donc possible. Dans PM², l'exclusion de transfert reste valide durant le *yield* ; il faut donc modifier le fonctionnement même de PVM pour autoriser un fonctionnement fil-synchrone des primitives. Dans Athapascan-0a, les réceptions se font à un moment bien déterminé (durant l'ordonnancement des fils d'exécution) ; l'exclusion est ainsi assurée sans nécessiter une opération particulière. La prise en compte d'un message entrant se fait soit lorsqu'aucun travail ne peut plus être effectué, donc avec un temps de réponse plus important mais avec un coût plus faible que pour TPVM / PM² ; soit à chaque commutation, ce qui revient sensiblement au même que les solutions de TPVM ou PM².

Dans PM², il y a un thread démon chargé de recevoir tous les messages (requêtes et réponses) et de les traiter. Ce démon est semblable au *master thread* de TPVM, et au code *watch-dog* d'Athapascan. L'implantation est optimisée dans le sens qu'un LRPC local est effectué spécialement, par partage de mémoire entre l'appelant et l'appelé. La mémoire partagée en question consiste tout simplement en les deux structures de paramètres d'entrée et de sortie du service. Selon l'application, le mode de description de données de PM² nous semble pouvoir être plus ou moins performant que celui d'Athapascan-0a. Si les données sont contiguës en mémoire, ou bien sont de petite taille, la description de PM² est probablement efficace, particulièrement pour un service local puisqu'aucun tampon de communication n'est employé. Par contre si les données sont de grande taille et dispersées en mémoire, le fait de préparer une structure de paramètres revient au même qu'emballer directement les données par PVM. A distance, PM² fait donc deux copies (une pour préparer la structure et une

pour l'emballer) de ce type de données alors qu'Athapascan-0a n'en fait qu'une (emballage direct) ; pour un service local, il y a le même nombre de copies et Athapascan-0a doit en plus faire un petit détour par PVM, qui comme PM^2 détecte que le message est local et le raccroche simplement à la file d'arrivée. L'implantation de PM^2 est aussi optimisée par un test d'hétérogénéité préalable à l'emballage. Le mode XDR de PVM n'est utilisé que si le site de destination utilise un encodage différent du site d'origine. Athapascan-0a pourrait employer le même genre d'optimisation (il ne le fait pas car la plupart des exécutions se font actuellement en mode homogène).

La portabilité de PM^2 ne nous semble pas très grande. En effet, et bien qu'il soit relativement facile de porter MARCEL, il est quasiment impossible de répercuter les modifications faites sur la version de PVM du domaine public sur une version propriétaire, comme PVMe. Athapascan-0a n'avait pas le but d'employer le temps partagé, et nous n'avons donc pas eu la nécessité de modifier PVM. Quoiqu'il en soit il nous semble que l'approche de TPVM, qui ne modifie pas non plus PVM et supporte cependant la préemptivité, est mieux adaptée. Hormis le fait que TPVM utilise une centralisation et une synchronisation excessives, nous ne pensons pas que son implantation de l'accès au réseau soit particulièrement moins efficiente que celle de PM^2.

PM^2 présente, mis à part l'extension de pile et la migration de processus légers, trois concepts intéressants : le retour d'erreur par levée d'une exception, le RPC sans retour et le mode *quick*. Athapascan-0a retourne actuellement toute erreur de façon fatale. Le RPC sans retour est une optimisation du schéma de décomposition procédurale parallèle, facile à implanter. Le mode *quick* existe aussi dans Athapascan-0a, mais n'est disponible qu'au niveau du noyau : on peut facilement, en compilant le noyau, ajouter de nouveaux points d'entrée systèmes qui s'exécuteront sans création d'un processus léger.

PM^2 est actuellement utilisé dans divers laboratoires, les applications privilégiées sont les applications irrégulières (optimisation combinatoire, méthodes hybrides itératives de convergence, support d'exécution de langages fonctionnels), les applications adaptatives (algorithmes génétiques, recherche tabou...) et les applications coopératives (support du compilateur de langage à objets distribués TOSCA, support d'applications coopératives...).

5.3 DTh

Distributed Threads[4] [Arbab 1994*c*] est le support d'exécution de l'environnement MANIFOLD [Arbab et al. 1993] développé au CWI d'Amsterdam. MANIFOLD est un langage de coordination pour la programmation d'applications parallèles sur des configurations hétérogènes. DTh s'appuie sur PVM et sur une interface pour la multiprogrammation légère.

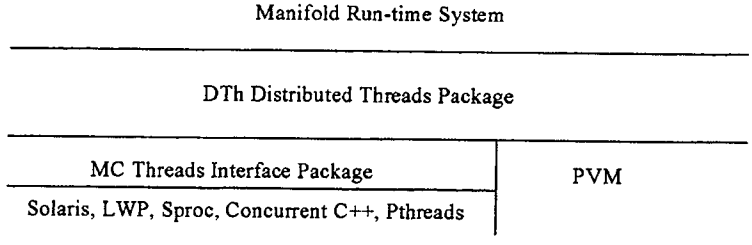

FIG. 5.11 - *DTh : organisation schématique.*

Le modèle de programmation de DTh définit les notions de tâche (processeur virtuel), de type de tâche (les différents binaires qui composent une application), de processus léger, de type de pro-

4. http ://cwi.nl/cwi/projects/manifold.html

cessus léger (la fonction de service), de moniteur et de type de moniteur. Il est possible d'instancier dynamiquement un type pour obtenir une nouvelle entité correspondante. DTh inclut trois concepts d'interaction : la création de processus léger à distance par appel d'une fonction de service (en fait c'est un RSR), l'échange de messages entre processus légers et les appels synchrones à des moniteurs gardés. Ce dernier concept est une sorte de super-RPC : un moniteur regroupe des données, des fonctions et des conditions (variables booléennes). L'appel d'une fonction d'un moniteur est synchrone et se fait en exclusion de toute autre fonction du même moniteur. Chaque fonction a des paramètres d'entrée et de sortie. Il est possible d'effectuer un appel « gardé » d'une fonction d'un moniteur en indiquant l'une des conditions du moniteur à l'appel. Dans ce cas la fonction ne sera exécutée que lorsque la condition sera vraie. Les données transmises (lors d'un RSR, d'un échange de message ou d'un appel de moniteur) sont typées, ce qui permet la gestion de l'hétérogénéité. Notons que les types possibles sont les types scalaires ou bien le type « bloc de mémoire » pour lequel les données sont transmises sans conversion. Il n'est donc pas possible d'établir une description de données complexe.

DTh doit posséder une connaissance globale des différentes entités manipulées, en particulier sur quelles tâches ces entités sont disponibles. Tout cela peut être définit manuellement, mais il existe un outil, *Build* [Arbab 1994*a*], qui permet, à partir d'une description de haut niveau, de générer le code de déclaration correspondant. Ce code déclare les différents types de tâches, les différentes fonctions de service et leurs profils, les différents types de moniteurs, leurs conditions, leurs fonctions et les profils de ces fonctions et la localisation des fonctions de service et des types moniteurs. Build s'occupe donc des définitions statiques. Un deuxième outil, *Config* [Arbab 1994*b*], fortement relié à PVM, permet le placement des tâches, leur identification croisée et le lancement de l'application. Cet outil utilise un fichier de configuration complexe pour connaître les différentes architectures, les différentes machines, les différents types de tâches, les placements possibles pour les nouvelles tâches...

Le modèle d'exécution de DTh n'est pas clairement décrit mais doit ressembler à celui de TPVM. Chaque processus léger possède sa propre boîte aux lettres. Un démon sur chaque tâche reçoit et tri les messages et sert les requêtes de RSR et d'exécution des fonctions de moniteur. Comme ce démon ne doit jamais se bloquer, toutes les fonctions de moniteur doivent être non bloquantes et en particulier ne pas utiliser la communication par envoi de messages ou bien faire des appels à d'autres moniteurs. Les processus légers « ordinaires » peuvent directement émettre sur le réseau ou bien communiquer localement en raccrochant leur message à la boîte du destinataire local. Les processus légers sont gérés en temps partagé, il doit donc y avoir une exclusion lors de l'accès au réseau. Une attente active en tourniquet semble être effectuée par les démons. L'efficacité de DTh ou sa disponibilité en tant que support exécutif indépendant ne sont pas l'objectif principal du projet MANIFOLD, par contre il semble être relativement largement porté. Comparé à Athapascan-0a, DTh présente un modèle étendu (moniteurs gardés et échange de messages entre processus légers). D'autre part les outils *Build* et *Config* n'ont pas d'équivalent ; dans Athapascan-0a le programmeur doit gérer lui-même ces fonctionnalités.

5.4 DTS

Distributed Task System[5] [T.Bubeck et al. 1995] est un support exécutif développé à l'Université de Tübingen, Allemagne. Ses buts sont premièrement le support sur machines à mémoire distribuée d'une bibliothèque de calcul formel, SAC-2. Cette bibliothèque fonctionne sur machines à mémoire partagée et suit un modèle *fork-join* dans lequel des processus légers effectuent les calculs parallèles. DTS permet son extension par la création de processus légers à distance, avec les limitations

5. http ://www-ti.informatik.uni-tuebingen.de/dts

suivantes :

- le grain doit être suffisant,
- la fonction calculée ne doit pas produire d'effet de bord,
- et les paramètres doivent pouvoir être copiés de façon transparente.

Le deuxième but de DTS est de montrer que le modèle de programmation *fork-join* est utile pour les applications irrégulières. Le modèle *fork-join* de DTS correspond à un RPC asynchrone ; il peut être traduit de différentes façons par l'exécutif en fonction du grain de calcul :

- soit comme un appel de procédure effectué par celui qui fait le *join* (grain fin),
- soit comme la création d'un processus léger local (grain moyen),
- soit enfin comme la création d'un processus léger réseau (gros grain et respect des limitations ci-dessus).

Le modèle de programmation de DTS est SPMD. Les tâches fonctionnent en mode client- serveur généralisé. Le mécanisme d'emballage des paramètres doit être fourni par l'utilisateur.

DTS régule la charge à l'aide d'une création paresseuse des tâches : seul le haut de l'arbre d'exécution est créé sous forme de processus légers. Une tâche particulière (LBP, *Load Balacing Process*) reçoit toutes les requêtes de création de processus légers réseau et les gère comme un *pool* de requêtes. Cette tâche connaît la charge (le nombre de requêtes en cours) de toutes les autres tâches. Lorsque cette charge devient insuffisante (les auteurs précisent que le nombre de 5 activités par tâche est suffisant pour obtenir une bonne régulation), une requête est sortie du *pool* et exécutée. Dans chaque tâche, un *node manager* sert les requêtes entrantes, ainsi que les ordres d'envoi de paramètres, les réceptions de résultats, et les complétions d'appel (réveiller ceux qui attendent sur un *join*). Chaque tâche possède aussi un processus léger *heartbeat* qui permet détecter si elle disparaît à cause d'une panne. Les activités qui s'y exécutaient sont alors redémarrées sur une autre tâche et les orphelins massacrés. Il est ainsi possible d'ajouter et de supprimer des machines durant l'exécution d'un programme.

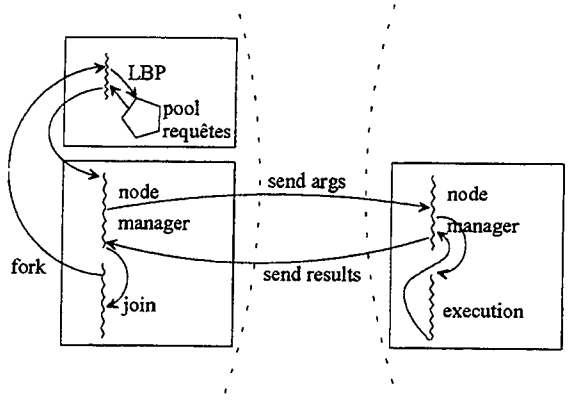

FIG. 5.12 - *DTS : modèle fork/join avec régulation de charge.*

DTS fonctionne sur quelques architectures, à l'aide de noyau de multiprogrammation préemptifs comme Solaris ou les C-threads, et de PVM. L'implantation exacte n'est pas décrite dans la littérature. Les auteurs de DTS rapportent qu'ils obtiennent de bonnes accélérations même pour des activités de grain relativement fin et qu'en conséquence DTS représente une combinaison attractive entre abstraction et efficacité pour des applications parallèles irrégulières. Le modèle de DTS est celui le plus proche d'Athapascan-0a ; c'est un modèle *fork-join*. Contrairement à Athapascan-0a, la régulation de charge est intégrée dans DTS ; d'autre part les exécutions distantes sont probablement de grain plus important que dans Athapascan-0a, qui n'effectue pas un tri comme dans DTS.

5.5 LPVM

LPVM (*Lightweight-process PVM*)[6] [Zhou & Geist 1995*a*], développé à l'ORNL vers 1995, vise l'expérimentation de PVM sur machines SMP. Sur ces machines, une implantation spécifique de PVM peut être envisagée, mettant à profit la multiprogrammation légère native pour implanter les tâches PVM de façon plus efficace que sous forme de processus lourds distincts et utilisant la mémoire partagée pour réaliser les communications entre tâches. Les questions principalement abordées sont :

- est-il possible d'améliorer les performances avec cette approche ?
- que doit-on modifier dans PVM pour qu'il supporte la multiprogrammation légère ?

LPVM ne fonctionne actuellement que sur une unique machine SMP. L'extension à des grappes de SMP est envisagée. LPVM a été développé à partir de l'implantation de PVM utilisant la mémoire partagée et un processus lourd par tâche (cette implantation est rapidement décrite en 3.2.3.5). Le démon est conservé tel quel (monoprogrammé), mais une application PVM est transformée en un unique processus lourd multiprogrammé. Chaque tâche PVM est réalisée par un fil d'exécution séparé. Le modèle de programmation est le même que celui de PVM : la mémoire partagée par les fils d'exécution n'est pas visible, la communication se faisant uniquement par échange de messages entre les tâches (fils d'exécution). L'ordre *pvm_spawn* de PVM, pour démarrer une nouvelle tâche, est implanté comme la création d'un nouveau fil d'exécution.

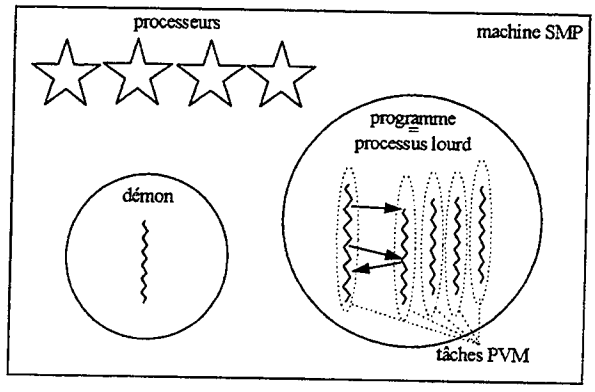

FIG. 5.13 - *LPVM : représentation schématique.*

Un programme est un unique processus lourd, chaque tâche du programme est réalisée par un fil d'exécution différent. La mémoire partagée n'est pas visible, les communications se font par échange de messages.

L'API de PVM n'est pas directement sauve vis à vis des processus légers puisqu'elle suppose un certain nombre de variables globales comme les identités des tampons de communication, leurs listes de fragments... La solution retenue pour la rendre sauve est d'ajouter l'identité du processus léger et l'identité du tampon de communication à chaque appel de primitive. LPVM présente donc quasiment la même API que PVM, ce qui rends la conversion d'applications très facile. A notre avis, le choix d'ajouter ces paramètres est inutilement lourd, puisqu'il est possible de cacher ces informations dans une zone de données spécifiques au processus léger concerné (tout comme on le fait habituellement pour *errno*).

L'implantation utilise les processus légers de niveau noyau. Elle a été réalisée à partir de la version utilisant la mémoire partagée de PVM, mais pourrait aussi être réalisée sur la version utilisant les *sockets*. Les modifications apportées à la bibliothèque PVM pour être sauve vis à vis des processus

6. http ://www.epm.ornl.gov/~zhou/ltpvm/ltpvm.html

légers correspondent grosso-modo à répliquer les données globales autant de fois que nécessaire pour que chaque processus léger gère ses propres données. De façon plus détaillée, il a fallu :

- « extérioriser » les identités des objets manipulés, c'est à dire les rendre facilement accessibles,
- identifier et dupliquer les données globales, qui sont maintenant gérées sous forme d'une liste avec un nœud pour chaque processus léger.

Un processus léger recherche ses données dans cette liste en fonction de son identité. Il retrouve alors sa propre image des données globales. Mis à part la gestion de cette liste, il n'y a pas besoin de verrouillage entre les processus légers, en particulier en ce qui concerne la gestion des messages, puisque chaque processus léger évolue dans « son propre espace ».

La primitive *pvm_spawn* est réalisée par le processus, sans intervention du démon, comme la création d'un nouveau processus léger. Cependant la primitive *pvm_mytid*, exécutée pour initialiser la bibliothèque PVM, enregistre la nouvelle tâche auprès du démon, comme avant. L'interaction avec le démon n'est donc pas éliminée pour autant. La primitive *pvm_spawn* ne spécifie plus le nom d'un programme à lancer, mais l'adresse d'une fonction à exécuter, puisque tout le code de l'application réside maintenant dans le même processus.

Les portages concernent Solaris 2, Aix 4 et Convex MPP. Les auteurs comparent l'efficacité à celle de PVM. En particulier, le temps de *spawn* est mesuré comme 10 à 20 fois plus court. La durée d'emballage des données est plus grande, ce que les auteurs n'expliquent pas puisqu'il n'y a pas plus d'exclusion. Enfin, le coût d'envoi est 50% plus important pour les petits messages, mais est inférieur à celui de PVM originel pour les grands messages. Ces mesures ne sont pas commentées.

L'utilisation principale de cette réalisation semble être le prototypage. En particulier, nous n'avons pas trouvé dans la littérature de comparaison de haut niveau (sur les applications). Les évolutions envisagées semblent être les grappes de SMP et l'intégration à une future version PVM 3.4. La principale conclusion de cette expérience nous semble être celle-ci : PVM est un modèle de programmation à mémoire distribuée ; sans « casser » ce modèle, il est difficile d'obtenir de grandes améliorations des performances sur une machine à mémoire partagée. LPVM est très spécifique au fonctionnement sur une machine SMP. A l'inverse, Athapascan-0a ne tire pas directement profit d'une machine SMP ; il faut concevoir un mécanisme de partage de mémoire pour l'utiliser efficacement dans ce cas.

5.6 DTMS

Distributed Task Management System[7] [Colin 1995*b*, Colin 1995*a*, Colin et al. 1995] est un environnement pour la programmation distribuée supportant un grand nombre d'activités parallèles de grain variable et régulant leur charge. L'un des buts est en particulier d'étudier l'influence de la granularité sur le comportement global d'une application. DTMS est développé à la Faculté Polytechnique de Mons, Belgique par J-N. Colin. DTMS se base directement sur les *sockets* d'Unix [Stevens 1990] et les processus légers de Solaris et d'OSF/1. Il est porté actuellement sur machines Sparc et Dec Alpha.

Le modèle de programmation de DTMS définit des sites (les machines physiques), des modules (les processus), des « fonctions comportementales » (les fonctions de service), des processus légers (l'auteur les appelle tâches mais nous les appellerons ici activités pour ne pas faire de confusion avec la notion de processeur virtuel dénommée « tâche » dans Athapascan-0a) et des messages. Toutes ces entités sont gérées dynamiquement. Ce modèle permet la création à distance de nouvelles activités (sans paramètres, unaire, synchrone) et l'échange de messages entre activités par l'intermédiaire de boîtes aux lettres. Les messages sont construits ou lus depuis des tampons de communication ; il

7. http ://pip.fpms.ac.be/~jnc

n'y a qu'un seul tampon de communication disponible par activité et par sens de communication : il n'est pas possible de préparer plusieurs messages concurremment. Il y a enfin trois politiques de régulation de charge, qui interviennent par le choix du site de démarrage d'une activité : deux en aveugle (aléatoire et cyclique) et une ne prenant en compte que la charge locale (par un seuil) et aveugle à distance (distribution aléatoire ou cyclique).

L'implantation reproduit en quelque sorte les fonctionnalités de PVM en ce qui concerne la gestion d'une machine virtuelle et les communications. Les communications sont réalisées à l'aide de TCP [Postel 1981*a*]. Un démon de communication est présent sur chaque site DTMS. Il brasse les messages et les retransmet au module destinataire s'il est sur le même site ou bien au démon de communication du site distant. L'emploi de démons de communication a la même finalité que pour PVM : réduire le nombre de canaux de communication. En complément de ces démons de communications il y a sur chaque site un processus serveur qui a une double fonctionnalité : conserver une réplique du « dictionnaire des fonctions » qui mémorise sur quel module une fonction peut être appelée et exécuter une version du « gestionnaire de charge » qui tient à jour la charge du site. Les processus serveurs gèrent donc des informations d'une façon distribuée, avec une cohérence faible et une mise à jour périodique. Comme pour PVM, il est possible d'agréger un nouveau site distant (d'y démarrer un démon de communication et un processus serveur), de le relâcher, de démarrer un nouveau module sur un site et de communiquer des messages. La différence principale avec PVM concerne la présence de multiples activités dans un module.

Les systèmes cibles sont considérés par l'auteur comme non saufs vis à vis des processus légers (Solaris jusqu'à la version 2.3 n'était effectivement pas « *MT-safe* » en ce qui concerne les *sockets*), aussi chaque activité accède le réseau en exclusion. En particulier, un gérant de module récupère tous les messages et les stocke dans les boîtes aux lettres destinatrices. Une activité peut cependant émettre directement et les messages entre deux activités du même module ne passent pas par le réseau. Les différentes activités d'un module sont gérées en temps partagé ; les activités système (le gérant de module par exemple) sont placés à une plus haute priorité. Dans les démons et les processus serveurs, un processus léger différent est employé pour chaque canal TCP.

Comme pour PVM domaine public, la présence des démons introduit un nombre conséquent de copies des messages qui transitent entre modules. La création d'une activité est ralentie par l'interrogation du processus serveur du site, pour savoir où la fonction comportementale est définie et pour éventuellement réguler la charge. Elle est synchrone puisqu'il faut connaître l'identité de l'activité créée pour pouvoir communiquer avec elle. L'auteur indique l'utilisation de DTMS pour des réseaux de neurones artificiels et des systèmes multi-agents. DTMS est en quelque sorte une réécriture de PVM qui soit fil-sauve. Le projet APACHE a choisi d'utiliser simplement une bibliothèque de communication plutôt que d'en écrire une nouvelle et l'a intégrée à un mécanisme de multiprogrammation légère.

5.7 MPI-F

MPI-F [Franke et al. 1994, Franke 1994*b*, Franke 1994*a*] est une version propriétaire de MPI qui intègre des processus légers. Elle a été développée par H.Franke au IBM Watson Research Center. Ses buts sont bien évidemment de fournir une version efficace de MPI sur les machines IBM, notamment la gamme SPx, mais aussi d'expérimenter l'intégration des processus légers dans les communications et d'évaluer une version multiprogrammée de MPI.

L'organisation de MPI-F est la suivante : l'adaptateur réseau HPS [Stunkel et al. 1994] est projeté en mémoire dans le processus utilisateur. Une couche système, la « *pipe layer* », défini un canal de communication vers chaque processeur distant. La progression des communications se fait soit par

interruption lors de l'arrivée d'un message, soit lors d'appels explicites à la fonction *kike_pipes*, plus éventuellement lorsqu'un délai maximal s'est écoulé. Notons que l'utilisation des interruptions introduit un surcoût important dans la latence des messages et ce mode n'est pas activé par défaut. La progression des communications se fait donc normalement par scrutation explicite ou périodique. Au dessus de la *pipe layer*, l'exécutif MPI-F gère à la fois les messages MPI, une forme de messages actifs permettant entre autre d'effectuer des copies de mémoire à distance, des messages à destination de démons serveurs et enfin des requêtes de création de processus légers. Au dessus de cet exécutif est reporté une version dérivée de l'implantation MPI-CH d'Argonne [Doss et al. 1993], qui prend en charge les communicateurs, la description de données à la mode MPI et les communications collectives.

Le modèle de programmation utilise trois paradigmes : l'échange de messages entre processus lourds, mais fil-synchrone, l'accès mémoire distant (sans gestion de cohérence ni synchronisation) et l'appel de service distant sous trois variantes dites *quick*, *non- threaded* et *threaded.*

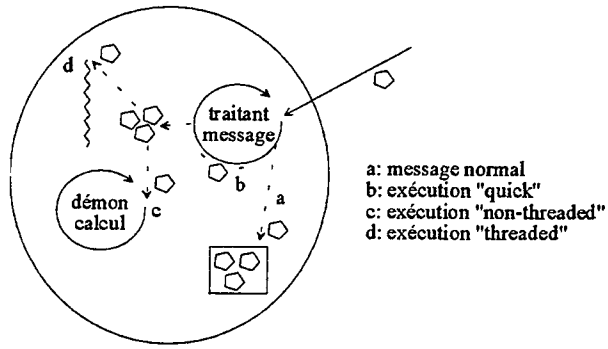

FIG. 5.14 - *MPI-F : gestion des messages.*

Un message entrant est récupéré par un traitant de message (analogue à un traitant ou gestionnaire d'interruption) et, selon son type, évalué différemment. Si c'est un message MPI normal, il est stocké dans l'ensemble des messages reçus que gère MPI. Si c'est un RSR quick, il est directement pris en charge par le traitant de messages. Si c'est un RSR non- threaded ou threaded, il est stocké dans un espace tampon en attente. Eventuellement, cet espace tampon sera visité, un message retiré et selon son type, le message sera évalué par un démon de calcul ou bien conduira à la création d'un nouveau processus léger.

L'implantation de MPI-F utilise des processus légers de niveau utilisateur, coopératifs, qui forment un sous-ensemble de POSIX. Le noyau de multiprogrammation des SP, DCE, a été dégrossi spécialement pour être plus efficace dans ces conditions. Dans la mesure où MPI-F n'est pas réentrant, les RSR en mode *quick* ne peuvent pas exécuter d'actions comme les appels à MPI ou aux processus légers, l'allocation de mémoire ou se suspendre. Un message de requête correspond à un entête MPI de taille fixe, un entête RSR de petite taille variable et des données de taille variable. Notons que MPI-F, lorsqu'il reçoit une requête, alloue la mémoire pour le descripteur et *pour les données*. En général il faudra donc recopier les données pour les placer à l'endroit où elles seront traitées.

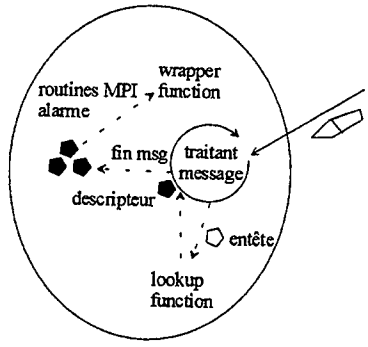

FIG. 5.15 - *MPI-F : le mode RSR threaded.*

MPI-F utilise deux rappels vers l'application lors de la réception d'une requête. Le premier rappel, vers la lookup function, est effectué lorsque l'entête du RSR est arrivé. La lookup function doit alors, à partir de cet entête, remplir un descripteur (situé en mémoire juste avant les données). Lorsque toutes les données sont reçues, le message est stocké dans l'espace tampon d'attente. Il n'en est sorti que lors d'un appel aux fonctions MPI ou bien sur un délai écoulé. Un deuxième rappel, la wrapper function est alors effectué si le RSR est de type threaded. Cette fonction doit, à partir du descripteur, traiter le message de la façon qui lui plaît.

L'API de MPI-F est quelque peu compliquée, puisque l'utilisateur doit compléter les définitions des structures qui donnent un format d'entête et un format de descripteur et doit aussi fournir les deux rappels cités précédemment. L'appel d'un RSR consiste à établir un entête RSR et à fournir des données décrites à la mode MPI.

MPI-F a été développé principalement pour les machines SPx. La version propriétaire IBM de MPI définitive est maintenant disponible, mais n'intègre pas les processus légers. La version MPI-F, utilisée pendant un temps par Nexus, n'est plus supportée par IBM. MPI-F est intéressant car il montre comment l'on peut coupler communication et multiprogrammation ; n'ayant pas accès aux mécanismes de base, le projet APACHE s'est restreint à une approche plus « pragmatique ».

5.8 Nexus

L'exécutif Nexus[8] [Foster et al. 1994*e*, Foster et al. 1994*a*, Foster et al. 1994*c*, Foster et al. 1994*d*, Foster et al. 1994*b*, Foster et al. 1996*c*, Foster et al. 1996*a*], développé au Mathematics and Computer Science Department, Argonne National Laboratory, vise à fournir et expérimenter un modèle d'exécutif parallèle pour applications irrégulières. Cet exécutif est prévu principalement pour servir de cible de compilateurs. Les buts recherchés, mis à part le support du parallélisme de tâche, sont le support de l'hétérogénéité de machines mais aussi de réseaux (protocoles de communication), l'efficacité, la portabilité et l'interopérabilité entre les compilateurs qui l'utilisent, permettant le développement d'applications hétérogènes (langages ou paradigmes de programmation variés).

Nexus définit cinq concepts principaux. Dans Nexus, un ensemble de *nœuds* est défini au lancement de l'application ; des nœuds peuvent être retirés ou ajoutés en cours d'exécution. Un nœud est seulement un ressource de calcul ; il n'a pas d'influence sur le réseau. Seul le nommage d'un nœud est dépendant de l'implantation. A l'intérieur d'un nœud, il y a plusieurs *contextes*, c'est à dire des espaces d'adressage, qui une fois créés ne peuvent pas être déplacés entre les nœuds. Un contexte ressemble à un processus Unix, sans la notion de protection toutefois. Un contexte est initialisé de façon

8. http ://www.mcs.anl.gov/nexus/index.html

synchrone à sa création ; il possède aussi une routine de nettoyage, activée lorsque sa destruction est demandée. L'initialisation et le nettoyage sont deux rappels vers l'application.

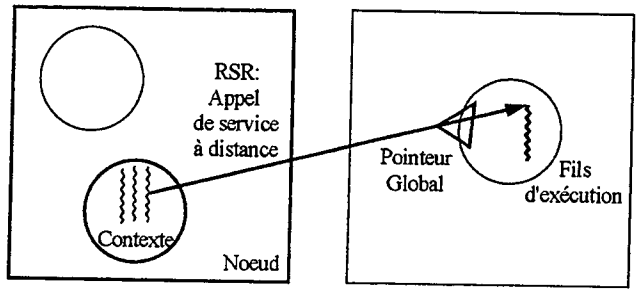

FIG. 5.16 - *Nexus : les concepts.*

Un contexte peut être par la suite activé par une *demande de service distant* (RSR) qui va créer un *processus léger* dans le contexte. Plusieurs processus légers peuvent exister concurremment dans un contexte, ils se partagent la mémoire disponible. Les fonctionnalités des processus légers sont les mêmes que celles définies par la norme POSIX ; complétées toutefois par la notion de RSR. Un **pointeur global** (*global pointer*) est l'information nécessaire pour identifier un nœud, un contexte sur ce nœud et une adresse locale dans ce contexte. Un pointeur global n'est donc pas transparent vis à vis de la localisation de l'objet pointé ; cependant, le pointeur global identifie l'ensemble des protocoles de communication par lequel l'objet peut être atteint (il contient en quelque sorte la référence de l'objet dans tous les protocoles possibles). De cette façon, Nexus autorise l'hétérogénéité de réseau de communication : à partir d'un site, pour accéder à un objet distant dont on connaît une référence (un pointeur global), il suffit de déterminer lequel, parmi les protocoles référencés dans le pointeur, est le mieux adapté. Cependant, la taille de chaque pointeur global est forcément importante, vis à vis d'un pointeur local ou de ce qu'il serait possible d'obtenir par un table d'adressage. De plus l'objet pointé ne peut pas être déplacé puisqu'il n'y a pas de transparence de localisation. Un pointeur global peut être transmis comme paramètre d'un RSR, permettant donc de manipuler des structures de données distribuées. Lors d'un RSR, le contexte destination est en fait identifié par un pointeur global.

Nexus s'appuie sur des mécanismes standard pour nommer les nœuds (protocoles de communication comme les sockets UNIX), les accéder (par rsh ou autre), accéder des fichiers répartis (NFS,...). Les protocoles de communication sont variés ; à la base seules des communications point à point sont nécessaires. Nexus utilise aussi bien des processus légers de niveau utilisateur que noyau, avec préemption ou sans. Notons que Nexus utilise des piles de taille fixe. L'interface de base des processus légers est POSIX, ou plus exactement PORTS0 (cf. 5.10). L'implantation de l'exécutif est variable en fonction du support disponible : La prise en compte des RSR entrants est soit non-préemptive (scrutation en fin d'exécution du processus léger ou lors de points de scrutation insérés dans le programme), soit préemptive (scrutation sur quantum de temps), soit *fil-synchrone* (mécanisme d'interruption matérielle lors de l'arrivée d'un message). L'idée est qu'il faut équilibrer le coût de la scrutation avec le temps de réponse, en tenant compte des fonctionnalités permises. L'API fournie par Nexus est de bas niveau ; elle met en avant les concepts présentés précédemment.

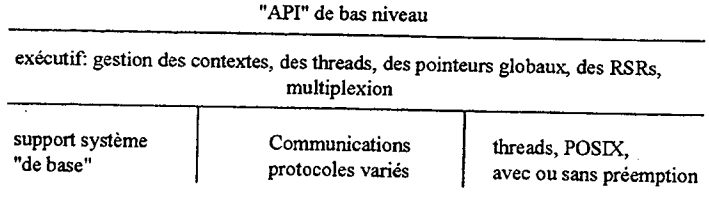

FIG. 5.17 - *Nexus : organisation schématique.*

Décrivons maintenant plus en détail le mécanisme de RSR. La référence du gestionnaire qui doit être activé par le RSR est une référence symbolique, mais un code de dispersion lui est associé, accélérant ainsi la recherche de l'adresse de la fonction à exécuter lors de la réception du RSR. Les gestionnaires de RSR sont identifiés par une déclaration, soit comme rapide (*non-threaded*), c'est à dire exécutés en série par un processus léger spécialisé à la manière d'un message actif, sans blocage, soit comme normaux (*threaded*), pour l'activation desquels un nouveau processus léger est créé. Le processus léger exécute alors la fonction déclarée, avec comme paramètre le pointeur global identifiant le contexte d'exécution et le tampon des paramètres du RSR. Les différents paramètres sont mis dans le tampon d'envoi au fur et à mesure (sous forme de blocs contigus d'un type de base). Notons que la sémantique est celle d'un transfert en place : les données ne doivent pas être modifiées tant que le RSR n'est pas envoyé. De plus, il est nécessaire de vérifier la place disponible dans le tampon d'envoi avant tout emballage de paramètre. Cette procédure nous semble un peu complexe, mais il est vrai que le code est normalement généré par un compilateur. Le site de destination étant connu avant l'emballage, il est possible de convertir la représentation des données, dès l'envoi, si les machines sont hétérogènes, et de ne pas convertir dans le cas contraire. Les paramètres sont déballés de la même façon qu'à l'emballage, sous forme de blocs contigus. Dans la mesure où un gestionnaire rapide peut avoir un accès exclusif à l'interface réseau, les primitives de lecture dans le tampon d'un gestionnaire rapide et dans celui d'un gestionnaire normal sont différentes. Il est possible qu'un gestionnaire rapide décide, pour continuer son traitement, d'activer un processus léger. Il doit alors transformer son tampon spécial en un tampon normal, accessible par le processus léger indépendamment des autres. De cette façon, on évite la copie de l'interface réseau dans un tampon général, pour une machine supportant la lecture directe dans l'interface réseau et les RSR vers des gestionnaires rapides. Notons que ce mécanisme est asymétrique : il n'est pas possible d'écrire de façon directe dans l'interface réseau, même si la machine supporte ce fonctionnement. Nexus spécifie explicitement qu'aucun appel n'est retardé indéfiniment, et que tous les appels d'un contexte sont exécutés dans l'ordre d'appel (même entre processus légers différents).

Nexus tourne sur beaucoup d'architectures (voir tableau des portages).De plus, il est possible d'appeler un dévermineur sur les contextes et de tracer l'exécution avec l'outil Pablo.

Machine - OS	Multiprogrammation	Communication (autre que UDP)
C90		
DEC AXP, OSF/1	QuickThreads	
HP9000, HPUX 9	DCE Threads	
IBM SPx	MPI-F	MPI-F
Intel Paragon, OSF/1	DCE Threads	Nx
ix86, FreeBSD	Provenzano Threads	
ix86, Linux	Provenzano Threads	
NeXTStep 3.0	C-threads (?)	
RS6K, Aix 3.2.5	DCE Threads	MPL, MPL+TCP
RS6K, Aix 4	Aix 4	
SGI IRIX 5.3, 6	QuickThreads	
Sparc, Solaris 2.3	Solaris	
Sparc, SunOS 4.1	FSU Threads, QuickThreads	ATM, Myrinet / UDP

TAB. 5.1 - *Nexus : les portages.*

Les expériences menées avec Nexus, relatées dans la littérature, montrent pour un SP2 avec la bibliothèque de communication MPL, 80% de surcoût dans la latence d'un ping-pong entre deux gestionnaires rapides et 30% de déficit de débit à mi-débit. Les expériences faisant intervenir des

communications entre processus légers sont bien sûr plus coûteuses, par exemple la latence est triplée et le débit à mi-débit est divisé par deux par rapport aux communications natives. Ces résultats ne sont pas étonnants, puisque sur le SP2, la création d'un processus léger coûte une bonne partie d'une latence de communication native (32μs de création plus 5.5μs de changement de contexte versus 44μs). Les auteurs cependant n'arrivent pas à comptabiliser toutes les occurrences de surcoût, sur une étude détaillée des temps. Ils attribuent la différence à un phénomène de cache (le test natif pouvant tenir dans le cache alors que le test de Nexus demanderais plus de mémoire). La scrutation multiprotocole aggrave le surcoût. Les auteurs précisent qu'il est nécessaire d'équilibrer les scrutations pour chaque protocole ; par exemple, scruter souvent un réseau rapide mais seulement rarement un réseau lent, de façon à minimiser le surcoût de scrutation sans trop réduire le temps de réponse vis à vis des réseaux.

Nexus est utilisé pour la réalisation de compilateurs pour des langages de haut niveau comme CC++ [Chandy & Kesselman 1993*a*, Chandy & Kesselman 1993*b*] ou Fortran M [Foster & Chandy 1994, Foster et al. 1993]. Il sert aussi de couche de base pour MPI-CH [Gropp et al. 1996], nPerl [Foster & Olson 1995], CAVEcomm [Cruw-Neira et al. 1993, Papka 1995]... Nexus est maintenant figé et ne devrait plus trop évoluer. La prochaine étape est **Globus** [Foster et al. 1996*b*, Foster et al. 1996*d*] : un environnement pour le calcul haute performance distribué. Cet environnement s'appuie sur Nexus et développe quatre axes principaux :

- la localisation des ressources : définir des mécanismes uniformes et extensibles pour identifier et localiser des ressources sur des machines distantes et les incorporer aux calculs en cours,
- la gestion des ressources et des protocoles : choisir les meilleurs moyens de calcul et de communication en fonction de leur qualité de service et de leur niveau de sécurité,
- l'accès aux données : développer des techniques pour fournir un accès uniforme, efficace et sécurisé aux fichiers
- la protection : inventer des mécanismes de délégation de responsabilité dans une large et changeante communauté répartie sur plusieurs domaines administratifs

Nexus est certainement plus optimisé qu'Athapascan-0a, en particulier sur le SP1. Comparé à Athapascan-0a, Nexus est de plus bas niveau et est moins simple d'emploi pour un programmeur. La notion de RSR mise en avant par Nexus permet l'implantation d'une décomposition procédurale parallèle, mais ne l'oblige pas. Cette approche est manifestement adaptée à l'intégration dans un compilateur.

5.9 Chant

Chant[9] [Haines et al. 1994*b*, Haines et al. 1994*c*, Haines et al. 1994*a*] est développé au ICASE, NASA LRC. Ses buts sont de définir une extension de POSIX pour le support du parallélisme de tâche qui soit :

- efficace (pas de copies des messages),
- portable (utilise les outils standards),
- utile (rajout de fonctionnalités)

L'organisation de Chant est un empilement de couches. Chant se base sur MPI et PORTS0 (cf. 5.10) pour définir une couche de communications point à point entre fils d'exécution. On ajoute ensuite les requêtes d'exécution à distance : mécanisme analogue aux RSR de Nexus, avec ou sans création de fil d'exécution. Au dessus, on trouve une couche « POSIX à distance » qui étends les fonctionnalités POSIX sur les processus légers à n'importe quel fil d'exécution à distance, notamment en ce qui

9. http://www.icase.edu/~haines/html/chant.html

concerne la création de processus léger, mais pas la synchronisation par verrou ou variable de condition. Une couche supplémentaire définit les Ropes [Haines et al. 1995*b*], des groupes de fils d'exécution et donc un espace de nommage et une étendue d'opérations collectives. Enfin vient une couche définissant des ShareD Abstractions (SDA) [Haines et al. 1995*a*], qui sont une sorte de moniteurs distribués. Cette dernière couche fait plus ou moins partie d'Opus [Mehrotra & Haines 1994, Chapman et al. 1994, Chapman et al. 1995], un environnement style HPF, mais étendu pour le parallélisme de tâches. Nous ne décrirons ni Opus ni les SDA ici. A chaque couche correspond donc un modèle de programmation différent.

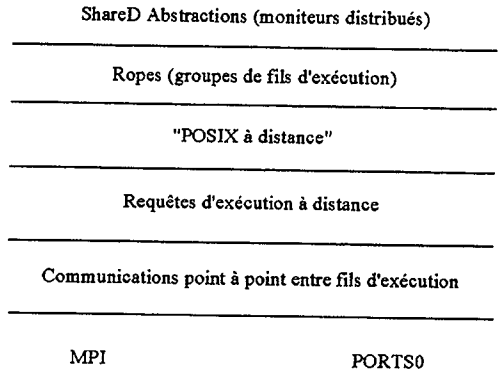

FIG. 5.18 - *Chant : organisation schématique.*

De même, à chaque couche correspond une API différente. L'échange de messages entre fils d'exécution présente une API réduite : envoi, réception, réception asynchrone (mais pas d'envoi asynchrone) de blocs de mémoire contigus, test et attente de fin d'opération asynchrone. Les requêtes de service à distance ont une API habituelle : enregistrement de gestionnaires, appel d'un service (un seul argument cependant). Les fonctions POSIX ressemblent à POSIX, le processus léger étant seulement identifié globalement à travers le réseau. Les fonctions Ropes permettent la création et l'attente de fin d'un Rope, la création de membres dans un Rope, l'envoi de messages à un membre (mais pas la réception comme il est expliqué plus loin), la diffusion, la synchronisation.

Les demandes sur le noyau de processus légers sont habituelles (création/suicide/yield/mutex), avec support de la préemption ou sans. Les demandes sur le noyau de communication sont l'échange de messages non bloquant (send / recv), la scrutation, les opérations collectives, la gestion de processus (rang, nombre de processus). L'identification des fils d'exécution est globale, formée d'une identification de contexte et d'une identification du fil d'exécution locale au contexte. La désignation n'est donc pas transparente et la migration de fil d'exécution n'est pas supportée. L'identification du correspondant lors de l'échange de messages se fait en partageant l'étiquette du noyau de communication entre identification du fil d'exécution destinataire et étiquette utilisateur. Il est donc impossible de filtrer selon le fil d'exécution source. Cela explique pourquoi l'écoute d'un membre particulier d'un Rope est impossible.

Les points d'entrée sont identifiés par des noms symboliques. Les RSR sont scrutés par un fil d'exécution serveur de plus haute priorité que les fils d'exécution de calcul ; le temps de réponse est donc petit. Les gestionnaires possèdent la particularité d'avoir comme paramètre, en plus de l'argument contenu dans le message, l'identification du fil d'exécution appelant. Il est ainsi possible de lui répondre. Notons qu'il existe deux moyens pour créer un fil d'exécution : soit la création à distance « POSIX », soit le RSR *threaded*. Ces deux moyens transmettent dans les deux cas un bloc d'arguments et nécessitent tous deux un enregistrement préalable du gestionnaire[10] .

10. dans l'hypothèse où les machines sont hétérogènes.

Les auteurs décrivent différentes implantations possibles pour la scrutation du réseau. La solution retenue semble être *thread poll* : le fil d'exécution qui communique est réveillé de temps en temps et scrute le réseau pour son propre compte. L'algorithme est alors le suivant :

```
attente d'un message(étiquette utilisateur) :
requête = irecv(tag construit à partir de l'identification du fil
            d'exécution et de l'étiquette utilisateur) ;
while (!test(requête) ) { yield; }
```

FIG. 5.19 - *Chant : scrutation directe.*

L'inconvénient de cette solution est de générer beaucoup de changements de contextes.

Les auteurs évaluent une autre possibilité dans laquelle un serveur de communication intervient. Le fil d'exécution qui attend une communication enregistre sa requête auprès du serveur et se bloque. Le serveur se réveille de temps en temps et test toutes les requêtes de communication en attente. Il réveille tous les fils d'exécution dont les requêtes ont abouties. L'algorithme devient :

```
attente d'un message(étiquette utilisateur) :
requête = irecv(tag construit à partir de l'identification du fil
            d'exécution et de l'étiquette utilisateur) ;
enregistrer la requête auprès du serveur de communication;
se bloquer;
```

FIG. 5.20 - *Chant : scrutation par appel d'un démon.*

L'algorithme du serveur de communication est :

```
pour toutes les requêtes en attente {
  if ( test(requête) == ok ) réveil du fil d'exécution demandeur
}
```

FIG. 5.21 - *Chant : démon de communication.*

L'inconvénient de cette solution est le fait qu'elle teste toutes les requêtes à chaque exécution du démon.

Les auteurs évaluent enfin une autre possibilité appelée *scheduler poll* : il s'agit d'équilibrer les commutations entre le serveur de communication et les fils d'exécution. Le serveur de communication est en fait l'ordonnanceur des fils d'exécution. Le fil d'exécution enregistre sa requête auprès du serveur et appelle l'ordonnanceur. Celui-ci va alors chercher un autre fil d'exécution à faire tourner. Si dans sa recherche il en trouve un en attente de communication, il effectue la scrutation pour son compte. Si le message est arrivé il relance le fil d'exécution, sinon il continue sa recherche. L'algorithme est le suivant :

```
attente d'un message(étiquette utilisateur) :
requête = irecv(tag construit à partir de l'identification du fil
            d'exécution et de l'étiquette utilisateur) ;
enregistrer la requête dans le TCB du thread;
yield;
```

FIG. 5.22 - *Chant : scrutation par l'ordonnanceur.*

L'algorithme de l'ordonnanceur est alors :

```
choix d'un thread prêt
s'il y a une requête dans son TCB
  et que test(requête) == nok : choisir un autre thread
sinon relancer thread
```

FIG. 5.23 - *Chant : ordonnanceur-scruteur.*

L'avantage de cette deuxième solution est qu'elle permet d'éviter de trop nombreux changements de contexte ou tests de requêtes. Cependant elle nécessite de modifier l'ordonnanceur.

Les auteurs mesurent que *scheduler poll* est seulement 10% plus rapide que *thread pool*. Ils acceptent donc une petite perte d'efficacité comme conséquence de la facilité d'implantation. La solution avec serveur de communication est systématiquement moins bonne que les deux autres.

L'implantation des Ropes nécessite un paragraphe à elle seule. La création d'un Rope n'est pas collective ; l'ajout et la suppression de membres est atomique. La création de fils d'exécution dans le Rope peut se faire durant la création du rope ou bien par ajout ultérieur de fils d'exécution. Un *Rope server* centralise les requêtes de création ou d'ajout de fils d'exécution et les dirige vers les bons contextes. Les opérations collectives sont faites d'abord localement, puis sont centralisées sur le *Rope server*. Les auteurs précisent qu'on ne peut pas utiliser les communications collectives de MPI, car elles sont bloquantes.

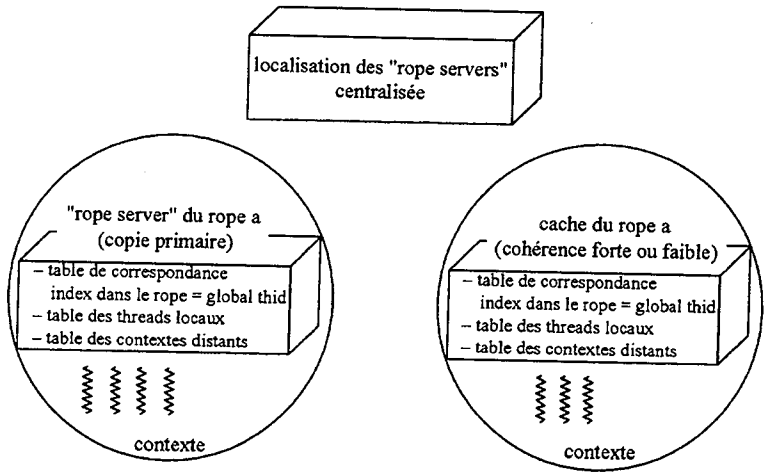

FIG. 5.24 - *Chant : structure d'un Rope.*

Les portages sont sur SUN avec P4 et Sun-LWP, Paragon avec Intel Nx et QuickThreads. MPI et PORTS0 sont aussi cités. Les tests d'efficacité, un ping-pong entre fils d'exécution préexistants, montrent un surcoût inférieur à 6% sans changement de contexte, et de 10 à 20% avec changement de contexte, sur des messages d'un Kilo-octets cependant. Les auteurs ne rapportent pas le surcoût sur la latence. L'utilisation de Chant se fait dans le cadre d'Opus et peut être de Vienna Fortran. Chant n'est pas disponible sur Internet au moment de la rédaction de ces lignes ; d'autre part les auteurs semblent plutôt s'orienter sur les SDA et Opus, c'est à dire les couches hautes, plutôt que sur l'implantation des couches basses. Chant part d'une extension de la multiprogrammation POSIX et propose un modèle complet (échange de messages entre processus légers, RSR, groupes, moniteurs...). Dommage qu'il ne soit pas largement disponible sur Internet.

5.10 Ports

PORTS[11] [PORTS Consortium 1995], Portable RunTime Systems, est un conglomérat de projets coopérants pour définir un support exécutif commun cible de compilateurs pour différents langages à parallélisme de tâches ou de données. Les membres sont les suivantes (par ordre alphabétique) :

- AWESIME, Univ. Colorado
- Chant, ICASE
- Chaos, Univ. Maryland
- Charm, Univ. Illinois, Urbana-Champaign
- Nexus, ANL, Caltech
- NPAC, Syracuse Univ.
- pC++, Indiana Univ, Univ. Oregon
- MPI-F, IBM Watson
- UCLA
- Vienna Fortran, Univ. Vienne

PORTS a vu le jour lors de SuperComputing'93. L'organisation de l'exécutif est en deux niveaux : un niveau PORTS0 dédié à la multiprogrammation et un niveau PORTS1 concernant la communication et les fonctionnalités annexes : horloges, événements.

Le niveau PORTS0 consiste en :

- une gestion de processus légers, "conforme" à un sous-ensemble du draft POSIX. Parmi les différences, citons :
  - il n'y a pas de *thread_join* (les processus légers sont toujours détachés)
  - il peut y avoir des rappels sur les créations/commutations/destructions de processus légers pour effectuer des prises de trace
  - il ne semble pas y avoir de politique d'ordonnancement définie.
- un sous-ensemble de la bibliothèque standard C réentrante et fil- synchrone, offrant les fonctionnalités suivantes :
  - la gestion dynamique de mémoire,
  - la gestion de fichiers (open, close, read, write, seek, fstat..)
- la définition d'un verrou de réentrance pour toutes les fonctions non- réentrantes,
- des fonctionnalités d'initialisation et de terminaison du paquetage et de récupération d'arguments sur la ligne de commande.

Le niveau PORTS1 consiste en :

- un concept de "modules mémoires" : avec en particulier leur identification (non transparente : numéro de nœud et numéro de contexte mémoire sur le nœud) et leur gestion (chargement, destruction).
- une notion de pointeurs globaux (référence à un module mémoire + pointeur local)
- différents paradigmes de communication (dont la définition est en cours) :
  - mémoire partagée : définition d'un gestionnaire d'allocation spécifique, puisque certaines machines ne peuvent partager que des zones mémoire spécifiques.
  - échange de messages "correspondants" (*matching message passing*) : à la base c'est un sous-ensemble de MPI, les communications se font entre processus légers ; les interactions entre les arrivées de messages et l'ordonnancement ne sont pas définies.

11. http ://www.cs.uoregon/paracomp/ports

  - accès mémoire à distance (*"network DMA"*) : les primitives de base sont les *Remote PUT/GET* asynchrones et des opérations de test et d'attente sont ajoutées.
  - demande de service distant (*Remote Service Request*, RSR) : entre contextes, le service est défini à l'aide d'un traitant de service, des paramètres peuvent être spécifiés, différents types d'exécution sont possibles (*quick, non- threaded, threaded,* cf. 5.7) .
- des mécanismes permettant l'utilisation d'horloges : l'implantation spécifiée doit être sauve vis à vis des processus légers mais ne doit pas les connaître (en particulier la possibilité d'avoir une horloge virtuelle par processus léger est considérée comme trop coûteuse dans un premier temps). Il y a deux horloges principalement : la *Very High Resolution Clock* (sa résolution est maximale, mais elle peut déborder rapidement) et la *Normal Wall Clock* (qui dure 12 heures au minimum, mais de résolution plus faible). Des primitives de création d'horloges, remise à zéro, démarrage, arrêt, lecture sont définies, ainsi que des informations sur la résolution, la durée de vie, la synchronisation des horloges. Les données sont stockées en format natif, puis converties en secondes en format flottant à chaque interprétation.
- des mécanismes de gestion d'événements : pour la prise de traces sous la forme d'une identité d'événement et d'un paramètre numérique.

Malgré le temps écoulé, la définition de PORTS n'est pas complète. Par exemple, la définition de PORTS0 n'est pas formelle ; la gestion de la préemption et des priorités n'est pas abordée, de même que le traitement des signaux ; il ne semble pas y avoir de fonctionnalité d'attente (comme *pthread_delay*)... PORTS1 n'est pas encore finalisé, l'idée à la base semble être de mettre le maximum de fonctionnalités de communication de façon à assurer l'efficacité sur tout type de machine. L'interaction avec MPI-2 n'a pas encore été abordée.

Seule une proposition pour les horloges et une implantation de PORTS0 sont disponibles (implantation réalisée par Argonne / Caltech). Cette implantation supporte les cibles suivantes :

RS6K Aix 3.2.5 / DCE Threads	HP/UX 9.x / DCE Threads
Sun OS 4.1.3 / FSU Threads (+GCC) / QuickThreads	Sun Solaris 2.3
NextStep 3.x	DEC OSF Alpha / Quick Threads
FreeBSD / FSU Threads (+GCC)	Linux / FSU Threads (+GCC)
IBM SPx MPI-F	

TAB. 5.2 - *PORTS-0 : les portages.*

Cette implantation est utilisée notamment par Nexus et devrait l'être à terme par tous les projets coopérants. L'évolution de PORTS a été jusqu'à présent très lente, à la fois parce qu'il s'agit d'un effort de coopération et par manque de crédits. La normalisation effectuée par PORTS n'est pas novatrice, il s'agit plutôt de définir un pot-pourri adéquat des concepts clés des différents participants. Il est dommage que l'équipe APACHE ne puisse faire partie de PORTS, par manque de crédits.

5.11 MPI-2

MPI-2 est une extension de MPI. Nous nous basons sur le brouillon du 27 novembre 1995 [Message Passing Interface Forum 1995]. MPI-2 n'intègre pas explicitement de processus légers, mais supporte la multiprogrammation. Parmi d'autres fonctionnalités, MPI-2 définit deux extensions qui nous intéressent ici.

La première concerne la copie de mémoire à distance. Dans MPI-2, une zone de mémoire typée est exportée collectivement et conduit à la création d'un nouveau communicateur. Toutes les fonctions

d'accès mémoire à distance sont effectuées en référençant ce communicateur. Les fonctionnalités disponibles sont :

- le *RemotePut* et le *RemoteGet* qui permettent l'écriture et la lecture à distance. Une double description de donnée doit être fournie : pour l'origine et pour la destination,
- l'*Accumulation* est analogue à un *RemotePut* mais modifie les valeurs déjà présentes à distance. Plusieurs fonctions d'accumulation sont disponibles ou peuvent être définies,
- le *Read-Modify-Write* effectue atomiquement une lecture, une modification et une écriture,
- la *Barrier* permet une synchronisation collective cohérente vis à vis des actions de mémoire distante.
- le *Fence* permet d'attendre la complétion à distance, pour un unique processus donné, des toutes les actions de mémoire distante.

La deuxième extension concerne la possibilité de définir des gestionnaires de communication ; la primitive *Post_handler* associe un traitant à une complétion de communication ; ce traitant est exécuté par le processus léger qui effectue le *Post_handler*. Il est possible d'assurer une exclusion mutuelle entre le gestionnaire et les autres processus légers. Cette extension simple permet d'effectuer deux opérations différentes : d'abord il est facile de réaliser de cette façon l'envoi en tâche de fond d'un ensemble de messages ; ensuite cela autorise la programmation d'un mécanisme de réception par interruption. Dans ces deux cas, le traitant n'a qu'à poster une nouvelle communication asynchrone et s'y attacher comme traitant de complétion. Un mécanisme de réception par interruption est la base idéale de tout exécutif mariant multiprogrammation légère et communications. Cependant, la spécification ne précise pas le coût de ce mécanisme implanté dans MPI-2, ni son temps de réponse, ni non plus les actions possibles dans le traitant. L'intérêt de MPI-2 pour la réalisation d'un exécutif pour applications parallèles irrégulières dépendra donc des caractéristiques d'une implantation particulière.

5.12 MPI-CH/Nexus

La version de MPI développée par Argonne [Doss et al. 1993] est maintenant portée au dessus de Nexus [Gropp et al. 1996]. Nous décrivons ici en quelques lignes les fonctionnalités de cette nouvelle version.

Nexus fournit des fonctionnalités de multiprogrammation ; dans MPI-CH/Nexus les processus légers ne sont pas visibles, les communications se font entre processus lourds. Les mécanismes d'identification et de filtrage sont les mêmes que ceux de MPI. Contrairement à Chant par exemple, il est possible de filtrer la réception par le processus origine. L'implantation de MPI-CH/Nexus est complètement fil-synchrone ; plusieurs fils d'exécution peuvent tenter de recevoir le même message (un seul réussira) et plusieurs communications collectives peuvent se dérouler concurremment, mais, et cela est dû au modèle de MPI qui ne permet pas l'identification des processus légers, un seul processus léger par processus lourd peut prendre part à chaque communication collective. MPI-CH/Nexus ne définit pas d'opérateurs nouveaux vis à vis de MPI ; gageons cependant qu'un version MPI-2 sera rapidement disponible.

Ce schéma « MPI multiprogrammé » ne nous semble pas, tel quel, facilement utilisable pour la programmation d'applications irrégulières ; le schéma de décomposition procédurale parallèle utilisé par Athapascan-0a nous semble plus structuré et donc plus approprié. Cependant, il est très probablement possible de reconstruire un schéma plus structuré au dessus d'un MPI multiprogrammé.

Diverses autres approches d'extension de MPI en relation avec la multiprogrammation légère sont décrites dans [Chowdappa et al. 1994, Skjellum et al. 1994*b*, Foster et al. 1995, Skjellum et al. 1994*a*].

5.13 Panda

Panda[12] [Bhoedjang et al. 1993] est le support exécutif du langage Orca [Bal et al. 1992] (langage à objets distribués et répliqués) en dehors du système Amœba [Mullender et al. 1990]. Panda est développé à la Vrjie Universiteit Amsterdam et au MIT. Le but de Panda est d'être une machine virtuelle portable offrant un support général et flexible pour implanter des exécutifs pour langages parallèles.

L'organisation de Panda est un peu atypique car elle fournit deux interfaces, l'une (*Panda Interface)* pour l'implantation du support exécutif du langage désiré et l'autre (*System Interface*) définissant un machine virtuelle portable. En conséquence Panda est divisé en deux parties, l'une (*Panda Layer*) indépendante de la machine et l'autre (*System Layer*) nécessaire pour porter Panda. Au minimum, les fonctionnalités du système cible sont la possibilité d'effectuer des envois de messages non fiables et de gérer des signaux lors de l'arrivée de messages ou lorsqu'un délai est écoulé. Panda a été implanté sur stations Sparc sous Unix.

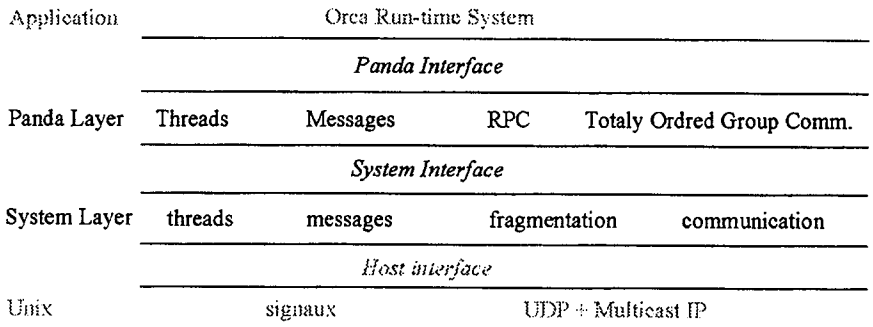

FIG. 5.25 - *Panda : organisation schématique.*

L'interface de Panda propose trois concepts. Les processus légers, qui peuvent être interrompus par l'arrivée d'un message mais qui ne sont pas gérés en temps partagé. Le RPC, qui est synchrone, fiable et utilise un serveur de noms pour trouver la localisation du service. Appeler un RPC correspond à la création d'un nouveau processus léger sur le serveur distant. Le troisième concept sont les communications de groupe totalement ordonnées [Kaashoek 1992]. Les groupes sont fermés. Le serveur de nom est aussi utilisé pour leur localisation. Il est possible de rejoindre ou quitter un groupe (en envoyant éventuellement un message à tous les membres du groupe pour les en informer). Il est bien évidemment possible de diffuser un message sur le groupe. Par contre il n'est pas possible de recevoir directement un message car les réceptions sont faites en tâche de fond par un (unique) gestionnaire établi préalablement. Les communications sont totalement ordonnées, ce qui signifie que tous les membres d'un groupe voient les messages dans le même ordre. C'est aussi là l'une des raisons de l'unicité du processus léger receveur. L'implantation utilise un séquenceur qui attribue un numéro à chaque message. Ce mécanisme facilite la conception de services distribués et est à la base du langage Orca.

L'interface système de Panda définit la même notion de processus légers, plus une notion de communication par échange de messages point à point et multipoints, sans étiquettes. La réception est faite un peu spécialement puisqu'elle fait intervenir un gestionnaire établi préalablement, qui est appelé de façon préemptive chaque fois qu'un message arrive. La description des données à envoyer est faite d'une façon particulière, qui permet la fragmentation du message et la réutilisation d'un entête commun à tous les fragments. Notons que l'interface système a été conçue pour être court-circuitée si le système de la machine cible fournit des équivalents. Par exemple il est possible d'utiliser

12. http ://www.cs.vu.nl/vakgroepen/cs/orca_papers.html

directement un noyau de multiprogrammation, ou bien une transmission de message *scatter-gather*, ou encore une diffusion multipoints matérielle. La mise en évidence de la fragmentation des messages permet de garder une bonne efficacité sur tout type de réseau. De même, la *Panda layer* prend en compte deux cas spéciaux pour optimiser son efficacité : la présence d'une communication multipoints fiable ou même ordonnée.

L'implantation sur Unix utilise les *pthreads* et UDP point à point ainsi qu'une extension d'IP : IP Multicast [Deering & Cheriton 91] qui permet d'effectuer des diffusions multipoints sur IP. Un démon de réception tourne à la plus haute priorité. Il effectue des entrée-sortie non- bloquantes. Si aucun message n'est présent il effectue une attente sur le signal SIGIO. Notons que l'implantation des *pthreads* utilisée définit un masque de signaux par processus léger.

L'intérêt principal de Panda réside dans sa volonté de définir une machine virtuelle portable. Cette approche n'est pas présente dans les autres supports exécutifs étudiés ; ils se contentent en général de considérer comme « standard » le noyau de communication et le noyau de multiprogrammation qu'ils utilisent. Panda est antérieur à Athapascan-0a ; nous n'avons pas utilisé Panda dans le cadre du projet APACHE. Les communications de groupe ne nous semblaient pas primordiales dans notre schéma de calcul ; d'autre part Panda ne dispose pas de fonctionnalités de gestion d'une machine virtuelle. Ces fonctionnalités offertes par PVM sont importantes dans Athapascan-0a.

5.14 Comparaison

Dans cette section nous allons établir une comparaison extensive des différents exécutifs présentés, en vue de dégager les points clés dans ce domaine. Notons que les éléments de cette comparaison sont tirés de la littérature citée ; dans certains cas nous n'avons pas pu déterminer avec précision telle ou telle caractéristique. Le point d'interrogation avoue alors notre ignorance. Une synthèse de cette comparaison est faite en 5.15.

Les différents points de comparaison retenus sont les suivants :

Nom	Emploi	Disponible	Evolue	Extension	Entités visibles
A0a	général, prototype	oui	non	non	service
TPVM	général, prototype	oui	oui	de PVM	*thread*, zone mémoire
LPVM	uniqt sur SMP, prototype	oui	oui (?)	de PVM	*thread*
PM^2	général	oui	oui	non	service
DTh	spécifique	licence	non (?)	non	service, *thread*, moniteur
DTS	spécifique	?	oui	non	service
DTMS	général	sur demande?	oui	non	*thread*
Panda	général	?	?	non	service, groupe
Nexus	général, cible de compilateur	oui	non	non	service
Chant	général	sur demande?	non (?)	"de 1003.1c"	thread, service, groupe
MPI-F	uniqt sur SPx, prototype	licence	non	de MPI	service, processus
MPI-CH/Nexus	général	oui	oui	de MPI	processus
MPI-2	spécification	pas encore (?)	oui	de MPI	processus, zone mémoire
Ports	spécification	PORTS0	oui	de MPI??	service, *thread*, zone mémoire

TAB. 5.3 - *Comparaison (1).*

- L'emploi : est-ce une spécification ou bien une implantation d'un exécutif d'emploi général ou spécifique à un environnement particulier, est-ce un prototype ?
- La disponibilité : disponible sur Internet, à travers une licence, ...
- La stabilité : la proposition est-elle stable ou bien évolue-t'elle encore ? est-elle maintenue ?
- La présentation comme une extension : l'exécutif est-il une extension d'un environnement ou modèle existant ou bien une entité nouvelle ?
- Les entités de programmation visibles : certains exécutifs purement fonctionnels ne présentent que la notion de service appelable à distance. Cette notion peut être étendue vers celle de moniteur. D'autres exécutifs présentent l'activité avec laquelle la communication aura lieu : soit le processus lourd (*processus*), soit le processus léger (*thread*), soit le groupe de communication. Enfin certains exécutifs introduisent la notion de zone mémoire accessible à distance.

Nom	Modèle Pt à Pt	Modèle Multipoints	Utilisation mémoire locale	Adm. processeurs	Adm. Contextes
A0a	RPC async, exclusif	ProcPar	oui	dynamique	dynamique
TPVM	Rcreate, MsgTh, Rmem	DataFlow, barrière	non	dynamique	dynamique
LPVM	Rcreate, MsgTh	aucun	non	1 seul SMP	dynamique (threads)
PM^2	RPC async, RSR	aucun	oui à travers Marcel	dynamique (?)	dynamique (?)
DTh	RSR, MsgTh, Moniteurs gardés	aucun	oui	dynamique, fich. de config	dynamique, fich. de config
DTS	RPC async	aucun	non	dynamique	dynamique
DTMS	Rcreate, MsgTh	aucun	non?	dynamique	dynamique
Panda	RPC	GroupesTO	oui	?	?
Nexus	RSR, GP	aucun	oui	dynamique (?)	dynamique
Chant	RSR, MsgTh	Ropes, OpCollTh	oui	?	?
MPI-F	RSR, Rmem, MsgProc	OpCollProc	oui	statique	1/proc
MPI-CH/Nexus	MsgProc	OpCollProc	oui à travers Nexus	statique (?)	statique (?)
MPI-2	IntRecv, Rmem, MsgProc	OpCollProc	indéfini	dynamique (?)	dynamique
Ports	RSR, MsgTh, RMem, ShMem	aucun	oui	non défini	non défini

TAB. 5.4 - *Comparaison (2).*

- Le modèle de communication point à point : trois grandes classes sont représentées : l'exécution à distance, la communication par échange de messages et l'accès mémoire distant. Chaque classe peut être découpée plus finement. Pour l'exécution à distance, il y a l'appel de moniteurs gardés, le RPC synchrone, le RPC asynchrone, le RSR (RPC sans retour) et la création de processus léger à distance (RPC sans retour ni paramètres, *Rcreate*). Il y a l'échange de messages entre processus lourds (*MsgProc*) ou légers (*MsgTh*), éventuellement complété par un mécanisme de réception par interruption (*IntRecv*). Il y a enfin l'accès mémoire à distance (*Rmem*), parfois doublé d'un mécanisme de partage de mémoire transparent (*ShMem*). Au minimum, une notion de pointeur global (*GP*) peut permettre de référencer des données à distance.
- Le modèle de communication multipoints : ce modèle est moins bien défini que le précédent. Certains exécutifs ne proposent qu'une restriction de communication à un groupe de participants (Ropes). L'ordre des communications peut éventuellement être totalement établit (GroupesTO). De même, certains exécutifs proposent explicitement la notion d'appel de procédures parallèles (ProcPar) ; la plupart à travers le mécanisme de RSR de plus bas niveau permettent l'implantation de cette technique sans l'offrir explicitement. Le vrai problème dans la communication multipoints concerne la synchronisation des activités. La barrière est le minimum, mais ne permet pas de communication de données. Certains proposent d'utiliser pour cela un modèle flot de données (*DataFlow*). D'autres utilisent la notion d'opération collective, les participants étant des processus lourds (OpCollProc) ou bien légers (OpCollTh).
- L'utilisation de la mémoire locale : est-il possible d'employer la mémoire locale d'une tâche pour communiquer entre activités locales ? Pour cela il faut que des primitives de synchronisa-

tion soient définies, soit explicitement, soit implicitement par l'exposition du noyau de multi-programmation.

- L'administration des processeurs : les processeurs sur lesquels l'application s'exécute sont-ils définis statiquement ou bien dynamiquement ?
- L'administration des contextes : certains exécutifs imposent un seul contexte par processeur physique, pour raison d'implantation (de contrôle d'accès à l'adaptateur réseau). La plupart offrent des contextes dynamiquement gérés, éventuellement à travers un fichier de configuration.

Nom	Définition globale des services	Talons	Création synchrone	Retour d'erreur	Identification des fils d'exécution
A0a	non	possibles	non	fatale	pas d'id
TPVM	oui	indirectement possibles	oui	?	contient la localisation
LPVM	non	indirectement possibles	oui	?	simple numéro
PM^2	non	obligatoires	non	exceptions	pas d'id
DTh	oui, typée	automatiques	oui	?	contient la localisation
DTS	SPMD	?	oui	non	pas d'id ?
DTMS	oui	indirectement possibles	oui	oui	contient la localisation
Panda	oui, serveur de noms	possibles	non	oui	pas d'id
Nexus	non	possibles	non	non	contient la localisation
Chant	non	possibles	oui	non ?	globale : contient la localisation, dans un groupe : serveur de noms
MPI-F	non	possibles	non	non	pas d'id
MPI-CH/Nexus	pas d'appels	indirectement possibles	pas d'appel	non	pas d'id
MPI-2	pas d'appels	indirectement possibles	pas d'appel	sans objet	pas d'id
Ports	non défini	non défini	non défini	non défini	non défini

TAB. 5.5 - *Comparaison (3).*

- La définition des services : certains exécutifs définissent localement leurs services appelables à distance. Le programmeur doit donc maintenir de lui-même la connaissance des possibilités de chaque contexte. Cette tâche est triviale si l'application est du type SPMD. D'autres exécutifs proposent une définition globale des services, éventuellement typée.
- La définition de talons : Les talons client et serveur d'un appel à distance peuvent être définis automatiquement ; ou bien il peut être obligatoire de les définir ; ou encore il peut être seulement possible d'en définir. Pour les exécutifs qui ne proposent pas la notion de service, elle peut être reconstruite à partir des mécanismes d'échange de messages. Les talons d'appel sont donc « indirectement possibles ».
- La création d'une activité : les exécutifs qui autorisent la communication ultérieure avec l'activité créée utilisent tous une création synchrone pour obtenir l'identifiant de la nouvelle activité. Les autres exécutifs emploient une création asynchrone, plus rapide.

- Le retour d'erreur : Lors de l'exécution distante, certains exécutifs offrent un retour d'erreur, soit par code d'erreur soit par un mécanisme d'exceptions. D'autres utilisent une reprise automatique. Certains ne proposent qu'un arrêt brutal du programme.
- L'identification des activités : nécessaire à la communication entre activités, l'identificateur peut contenir la localisation, ce qui limite fortement la possibilité de migration mais est simple à mettre en œuvre, ou bien ne pas la contenir, ce qui nécessite l'implantation d'un serveur ou d'un mécanisme de localisation.

Nom	Description des données	Filtrage des messages en réception	API
A0a	à la PVM	exclusion	C+macros
TPVM	à la PVM	selon tag et origine	lib C
LPVM	à la PVM	selon tag et origine	lib C
PM^2	à la PVM	aucun?	C+macros
DTh	file de types de base	selon tag et origine	lib C
DTS	?	?	lib C (?)
DTMS	file de blocs non typés discontigüs	selon tag et origine	lib C
Panda	file de blocs non typés discontigüs	aucun	lib C
Nexus	file de blocs typés discontigüs	aucun	lib C
Chant	1 bloc typé	selon tag uniquement	lib C
MPI-F	à la MPI	selon tag, communicateur et proc. origine	lib C
MPI-CH/Nexus	à la MPI	selon tag, communicateur et proc. origine	lib C, Fortran
MPI-2	à la MPI	selon tag, communicateur et proc. origine	lib C, Fortran
Ports	non défini	non défini	

TAB. 5.6 - *Comparaison (4).*

- La description des données : elle peut être à la MPI, c'est à dire une carte mémoire des données typées et discontiguës. Cette description est en générale « statique ». Une autre description est à la PVM, c'est à dire une file de blocs de données typées discontigüs, établie lors de chaque transmission. Ce mécanisme est dynamique, mais généralement plus lent que le précédent. Certains exécutifs ne proposent qu'un seul bloc ou un typage restreint voire inexistant.
- Le filtrage des messages : certains exécutifs permettent la sélection d'un message parmi l'ensemble de ceux qui ont été reçus. Cette sélection peut être effectuée selon une étiquette adossée au message, selon l'activité qui a émis le message ou selon le plan de communication sur lequel il transite. Certains exécutifs ne permettent qu'une réception dans l'ordre d'arrivée. Dans le cas d'une interaction client-serveur, une exclusion ou un mécanisme de gardes peut être fourni.
- L'API : c'est soit « un langage » formé de macro-instructions structurelles et de primitives, soit une simple bibliothèque de fonctions, pour le langage C ou Fortran.

Nom	Fonctions *threads* locales	Fonctions horloge	Fonctions fichiers	Ordonnancement global
A0a	non sauf synchro	interne : locale + globale	non	
TPVM	non	non	oui (fichier log)	
LPVM	non	non	non	
PM^2	non	non (?)	non	selon priorités
DTh	oui	non	non	
DTS	oui (?)	non	non	par régulateur
DTMS	non	non	non	
Panda	oui	non	non	
Nexus	oui	non (?)	non	
Chant	oui	non	non	
MPI-F	oui	oui (locale + globale)	non	
MPI-CH/Nexus	non	non	non	
MPI-2	non	non ?	non ?	
Ports	oui	oui (locale)	oui (non parallèles)	

TAB. 5.7 - *Comparaison (5).*

- La visibilité des fonctions de multiprogrammation légère : certains exécutifs exposent complètement le noyau de multiprogrammation légère. D'autres ne montrent que les fonctions de synchronisation, voire même interdisent l'emploi de la multiprogrammation légère en dehors des possibilités offertes par le modèle de programmation.
- Les fonctions d'horloge : certains exécutifs offrent des fonctions de prise de temps. D'autres ne les offrent que de façon interne à l'exécutif, et non dans le modèle de programmation. Les fonctions d'horloge peuvent être selon un temps local, propre à chaque processeur, ou bien synchronisé globalement entre tous les processeurs.
- Les fonctions de fichier : la plupart des exécutifs présentés n'offrent pas de fonctionnalités particulières concernant les fichiers. Certains offrent une notion de fichier centralisé. L'intégration avec des fonctionnalités de fichiers parallèles n'a pas encore été décrite.
- L'ordonnancement : l'ordonnancement est très rarement coordonné entre les processeurs. Cela reste une voie à explorer.

Nom	Outils annexes	Approche
A0a	environnement global, consoles texte et graphique, réexécution déterministe, traces, déverminage	Pragmatique
TPVM	console ?	Pragmatique
LPVM		Intégrée
PM^2	console ?	Pragmatique
DTh	console, déverminage, prog. visuelle	Pragmatique
DTS	tolérance aux fautes	Pragmatique
DTMS		Pragmatique
Panda		Idéaliste
Nexus	traces (pablo)	Intégrée
Chant		Pragmatique
MPI-F		Intégrée
MPI-CH/Nexus	comme Nexus ?	Pragmatique
MPI-2		
Ports		

TAB. 5.8 - *Comparaison (6).*

- Les outils annexes : certains exécutifs supportent des consoles, la prise de traces, la réexécution déterministe, le déverminage...
- L'approche : l'approche de conception peut être pragmatique, c'est à dire faire fonctionner côte à côte un noyau de communication et un noyau de multiprogrammation et définir ces noyaux comme prérequis de portabilité, ou bien intégrée, auquel cas le noyau de communication et le noyau de multiprogrammation interagissent fortement et l'implantation est généralement différente pour chaque portage. L'approche peut enfin être idéaliste, définissant une machine virtuelle comme interface de portabilité.

Nom	Base	Caractéristiques Base	Méthode d'implantation	Portabilité multiprogrammation
A0a	PVM	non fil-sauve	coroutines	grande
TPVM	PVM	non fil-sauve	exclusion globale + verrouillage API	grande
LPVM	PVM	non fil-sauve	réplication + verrouillage API	?
PM^2	PVM	non fil- sauve	verrouillage à l'intérieur	noyau Marcel
DTh	PVM	non fil-sauve	?	grande
DTS	PVM	non fil-sauve	accès exclusif	1003.1c
DTMS	TCP	fil synchrone?	kernel threads + verrouillage API	1003.1c, Solaris
Panda	UDP	non fil-sauve	?	1003.1c
Nexus	divers...	variable...	variable...	grande
Chant	Nx, P4, MPI	non fil- sauve	exclusion globale + verrouillage API?	1003.1c
MPI-F	pilote CSS-CI	non fil- sauve	?	non prévue
MPI-CH / Nexus	Nexus	fil synchrone	opérations bloquantes?	API Nexus
MPI-2				
Ports				

TAB. 5.9 - *Comparaison (7).*

- La base de communication : La base la plus courante est PVM. MPI commence à venir et bien sûr TCP/UDP ont été aussi utilisés. Certains exécutifs s'implantent directement sur un pilote d'adaptateur réseau.
- Les caractéristiques de cette base : elle peut être quelconque, ou bien sauve vis à vis des processus légers, ou encore fil-synchrone. L'implantation en dépend fortement.
- La méthode d'implantation : La méthode la plus simple est l'accès exclusif au noyau de communication. L'utilisation de coroutines est une méthode dérivée qui permet aussi de façon transparente un accès exclusif au noyau de communication. Un méthode plus évoluée, nécessaire pour le partage de temps entre activités, est l'emploi d'un verrouillage des données partagées et d'une synchronisation de l'accès au noyau de communication. Ces deux méthodes rendent le noyau de communication sauf vis à vis des processus légers. Une troisième méthode, permettant peut être une meilleure efficacité, est l'utilisation d'une exclusion fine à l'intérieur du noyau de communication. Plusieurs communications peuvent ainsi se dérouler en concurrence, le noyau de communication étant rendu fil- synchrone. Enfin, s'il possède déjà cette caractéristique, il suffit d'employer des opérations de communications bloquantes.
- La portabilité vis à vis du noyau de multiprogrammation : elle peut être grande si les demandes sur le noyau sont faibles. Elle peut être restreinte à un noyau présentant l'API de la norme Posix 1003.1c. Elle peut aussi demander un noyau spécifique.

Nom	Portabilité communication	Nbr de portages	Hétérogénéité	Recopies
A0a	API PVM	7	oui	oui / non à l'émission si contiguës
TPVM	API PVM	3	oui	oui / non à l'émission si contiguës
LPVM	impl. générique	3	oui (?)	probablement
PM^2	version PVM	5	oui	oui (?)
DTh	API PVM	>5	oui	oui (?)
DTS	API PVM	4	oui (?)	oui
DTMS	Unix, TCP	2, générique	non	oui
Panda	reconstruction SL	1, générique	non	pas obligatoirement
Nexus	impl. générique, adaptation	>13	oui	pas obligatoirement
Chant	API MPI	>2	oui (?)	pas obligatoirement
MPI-F	non prévue	1	non	non
MPI-CH / Nexus	API Nexus	autant que Nexus?	oui	non
MPI-2			oui	non
Ports			non défini	non défini

TAB. 5.10 - *Comparaison (8).*

- La portabilité vis à vis du noyau de communication : Elle peut être restreinte à un noyau offrant une API particulière, généralement PVM, MPI ou *sockets* Unix. Elle peut être encore plus restreinte par la nécessité d'utiliser une version spécifique d'un noyau. A l'inverse, elle peut être grande, par la prise en compte de plusieurs noyaux différents.
- Le nombre de portages : ce nombre est intéressant pour l'utilisateur qui a besoin d'un exécutif réellement porté.
- L'hétérogénéité : l'hétérogénéité de format de données entre machines peut être ou ne pas être prise en compte.
- Les recopies de données : certains exécutifs font obligatoirement des copies des données qui sont communiquées. D'autres arrivent à éviter les copies dans certains cas, ou même complètement.

Nom	Fonctions spéciales processus légers	Tailles des piles	Priorités	Temps partagé
A0a	instrumentation, réutilisation	fixe	non	non
TPVM		fixe	non	oui
LPVM		fixe	non	oui
PM^2	extension de pile, migration, instrumentation	variable, extensible	oui	oui
DTh		fixe	oui	oui
DTS		?	oui ?	?
DTMS		fixe	oui	oui
Panda		fixe ?	oui	non
Nexus		fixe	oui	possible
Chant		?	oui	possible
MPI-F	optimisation coroutines	variable possible	oui	non
MPI-CH / Nexus		pas d'appel	oui à travers Nexus	possible
MPI-2		pas d'appel	non défini	possible
Ports		non défini	oui	possible

TAB. 5.11 - *Comparaison (9).*

- Les fonctionnalités spéciales pour les processus légers : certains exécutifs en proposent comme l'instrumentation (combien d'activités prêtes ?), diverses optimisations, l'extension de pile, la migration...
- La taille des piles : elle est généralement fixe, mais peut être un paramètre de la création à distance. Dans certains cas les piles peuvent être extensibles.
- Les priorités : la plupart des exécutifs offrent un mécanisme de priorités différentes entre activités.
- Le temps partagé : certains exécutifs gèrent les activités en temps partagé, d'autres non. Certains enfin laissent le choix à l'utilisateur, ou bien fixent le comportement en fonction des capacités de la machine cible.

Nom	Scrutation Comment?	Scrutation Qui?	Scrutation Quand?
A0a	scrutation	démon puis ordonnanceur	sur actions utilisateur ainsi qu'à chaque commutation ou qd aucun prêt
TPVM	scrutation	thread	à son tour
LPVM	scrutation	thread	?
PM^2	scrutation	thread	à son tour
DTh	scrutation	démon	périodique (?)
DTS	scrutation	démon	?
DTMS	scrutation (?)	démon	?
Panda	interruption	démon	à la + haute priorité
Nexus	interruption ou scrutation	traitant ou démon ou thread (?)	variable...
Chant	scrutation	thread	à son tour
MPI-F	interruption ou scrutation	traitant	sur actions et périodique
MPI-CH / Nexus	comme Nexus	comme Nexus	variable...
MPI-2			
Ports			

TAB. 5.12 - *Comparaison (10).*

Les points suivants concernent la prise en compte des communications :

- Comment est fait cette prise en compte : en général l'interruption est trop coûteuse pour être utilisée. La scrutation est donc prépondérante. Certains exécutifs utilisent la meilleure des deux méthodes selon les caractéristiques des machines cibles.
- Qui prend en compte les messages entrants : cela peut être le traitant qui reçoit le message. Cela peut être un démon de communication. Dans ces deux cas, le processus léger destinataire est réveillé si nécessaire. Enfin, ce peut être le processus léger lui même qui effectue l'attente du message.
- Quand le message est-il pris en compte : lorsqu'il y a préemption, le message est pris en compte immédiatement. Lorsqu'il y a scrutation, le message peut être pris en compte de façon périodique (à la fin d'un quantum de temps), en fonction de l'exécution d'un processus particulier (le démon ou le processus qui attend le message) ce qui se produit généralement selon un tourniquet entre les différents processus (*à son tour*), ou lors d'actions effectuées par l'utilisateur, implicitement ou explicitement, ou encore à chaque commutation ou enfin quand il ne reste aucun processus léger prêt.

Nom	Mesures importantes
A0a	latence, débit, recouvrement des attentes de communications, applicatives (calcul symbolique, dynamique. moléculaire, régulation de charge)
TPVM	latence, débit, applicatives
LPVM	latence, débit
PM^2	latence, durée de migration, applicatives (combinatoire, régulation de charge)
DTh	?
DTS	applicatives (calcul symbolique)
DTMS	latence, débit
Panda	latence, débit
Nexus	latence, débit, recouvrement des communications, multiprotocole
Chant	latence, débit, applicatives
MPI-F	latence, débit
MPI-CH / Nexus	applicatives
MPI-2	
Ports	

TAB. 5.13 - *Comparaison (11).*

- Les mesures importantes : nous essayons ici de regrouper les différentes évaluations qu'ont utilisées les auteurs. En général, ce sont des mesures de bas niveau (latence, débit comparé au noyau de communication de base) ou applicatives. Peu d'exécutifs ont été évalués en ce qui concerne le recouvrement des communications et l'ordonnancement.

5.15 Conclusion

En conclusion de cette étude de différents exécutifs qui ont tous un but commun, le support du parallélisme de tâches au dessus d'un système d'exploitation classique, nous apercevons, au delà de leurs différences, une organisation générale.

Cette organisation est l'articulation d'un certain nombre de composantes :

- un noyau de multiprogrammation,
- un noyau de communication,
- des fonctionnalités pour la gestion de contextes d'adressage,
- des fonctionnalités pour la gestion d'un stockage temporaire (fichiers),
- des fonctionnalités pour l'interaction avec l'utilisateur (terminal),
- des fonctionnalités pour la gestion du temps (horloges).

Chaque composante a sa propre gamme de diversité. Indépendamment des différents choix possibles, les exécutifs présentés ont tous mis l'accent sur les fonctionnalités de multiprogrammation et de communication et l'interaction entre ces deux noyaux. Les autres composantes sont considérées de façon plus annexe, les solutions pratiquées dans les systèmes d'exploitation en général étant applicables. Notons que la notion de gestion de fichiers parallèles n'est pas intégrée actuellement dans les noyaux présentés. L'interaction entre un système de fichiers parallèles et un noyau de multiprogrammation n'est pas clairement définie, bien que certains travaux soient en cours [13] .

13. voir notamment http ://www.mcs.anl.gov/home/itf/sio-threads.html

Les noyaux de multiprogrammation retenus sont, à la base, semblables à la définition Posix. Cela principalement à cause de la disponibilité de tels noyaux. Cependant, deux grandes lignes d'exploration existent : la première concerne les coûts, qu'il est possible de réduire sous certaines conditions d'utilisation. Par exemple nous avons montré que la réutilisation de fils d'exécution pouvait être intéressante, sous la condition de conserver une taille de pile identique. De même, il est possible d'optimiser la commutation de contexte en limitant le nombre de registres à sauvegarder, par exemple en excluant ceux du coprocesseur mathématique si le fil d'exécution ne s'en sert pas. Une autre optimisation possible est de simplifier l'exclusion mutuelle par l'emploi du coroutinage. Cela est en particulier effectué par MPI-F, qui réduit ainsi le temps d'acquisition d'un verrou.

La deuxième voie d'exploration concerne les extensions possibles et utiles de la définition Posix. Nous avons vu que des informations de charge, en particulier le nombre de fils d'exécution prêts, peuvent être utiles. De même, des rappels lors de certaines actions seraient utiles pour compléter la prise de traces. Certains rappels sont prévus dans Ports. Une troisième possibilité d'extension est l'exposition de l'auto-ordonnanceur. Actuellement l'auto-ordonnanceur est opaque, avec deux modes généraux de fonctionnement : temps partagé et fifo prioritaire. Principalement à cause de l'interaction avec le noyau de communication, l'ordonnanceur des fils d'exécution devrait être commandable plus finement : par exemple pour prendre en compte un message le plus rapidement possible. Ces fonctionnalités serviraient pour définir un auto-ordonnancement non pas local, mais global, comme par exemple celui basé sur les priorités exploré par l'équipe ESPACE et l'auto- ordonnancement par bande exploré récemment par l'équipe APACHE [Marchal 1996]. Mis à part ces deux voies d'exploration, un certain nombre de questions annexes, la plupart déjà posées, subsistent. Citons par exemple l'interaction avec le système (les processus légers de niveau noyau), celle avec les traitants d'interruption, l'utilisation du vrai parallélisme, la gestion des piles, la préemption, la nécessité d'avoir une *libc* réentrante portable...

Nous avons vu que les noyaux de communication sont assez diversifiés. Cela vient en partie du fait que, contrairement aux noyaux de multiprogrammation, il n'existe pas de standard réel. La norme MPI existe, mais l'échange de messages n'est pas la seule méthode de communication utilisable. Nous avons montré différentes notions « d'entité avec qui l'on correspond » et les mécanismes de communication associés :

- la procédure (le RPC synchrone ou asynchrone),
- le traitant (le RSR),
- le port (communication par flot, non présent dans les exécutifs décrits),
- le module mémoire (accessible par *Network DMA*), qui peut se concevoir comme un véritable *objet*,
- l'espace mémoire global (utilisation du partage de mémoire),
- le processus léger (messages entre fils d'exécution),
- le processus lourd (messages entre processus).

L'un des choix que doit faire le concepteur d'un exécutif parallèle est la mise à disposition d'un seul ou de plusieurs de ces mécanismes. C'est un choix entre simplicité de l'outil et efficacité, puisque selon les applications et les plates-formes cibles, tel ou tel concept sera plus performant. L'activité avec qui l'on correspond peut être créée lors de la communication ou bien exister préalablement. Par exemple, appliqué au concept de procédure, on obtiendra soit l'appel de procédure à distance soit le rendez-vous. En cas de création, elle peut être unilatérale (à la RSR), ou bien collective (à la *dataflow* par exemple). La création peut être paramètrée de façon plus ou moins complexe. La synchronisation ultérieure entre le créateur et le créé peut être déterminée dès la création, ou bien faire l'objet d'un protocole postérieur. L'identification de la nouvelle entité peut contenir la localisation (cas le plus courant) ou bien un mécanisme de localisation sera mis en jeu.

L'efficacité des communications est bien évidemment cruciale. Cela passe par une minimisation des copies de données mais aussi par une minimisation du coût de l'algorithme de contrôle (en particulier, employer le moins de messages possible pour la réalisation de cet algorithme). Les propriétés de base (fiabilité, ordre, équité, contrôle de flux) sont centrales. L'existence de plusieurs contextes de communication indépendants est utile pour la réalisation de bibliothèques de fonctions réutilisables. La réception peut nécessiter de traiter les messages comme ils arrivent, ou à l'inverse permettre un filtrage sur le contexte, l'origine ou l'étiquette du message. Il peut y avoir des communications collectives (en particulier diffusion, synchronisation...). La dernière question concernant les communications est la description des données qui sera employée : est-ce une carte mémoire à la MPI ou une liste de données à la PVM ? Cette description doit être assez souple pour prendre en compte l'aspect dynamique des données manipulées par les applications irrégulières, sans introduire des surcoûts de copie.

L'interaction entre noyau de communication et noyau de multiprogrammation peut être envisagée des deux côtés. Nous avons vu que la progression des communications peut se faire soit par interruption sur arrivée d'une communication, soit par différentes méthodes de scrutation (périodique, sur action de l'ordonnanceur, sur action de communication, sur des points de scrutation insérés par l'utilisateur). De même, la progression des calculs peut être préemptive (sur temps partagé ou uniquement sur entrées-sorties), ou bien coopérative (c'est à dire sur une action d'ordonnancement implicite ou explicite (*yield*)). La question fondamentale de l'interaction entre noyau de communication et noyau de multiprogrammation est la liaison entre ces deux progressions.

Il est possible d'assujettir la progression des communications à l'auto-ordonnancement des fils d'exécution, par exemple en scrutant le réseau lors d'actions d'ordonnancement, cette scrutation pouvant être faite comme nous l'avons vu soit par le fil d'exécution qui communique, soit par un démon, soit par l'ordonnanceur lui-même. A l'inverse il est possible d'assujettir l'auto-ordonnancement des fils d'exécution à la progression des communications. C'est à dire de réordonnancer lors de l'arrivée d'une communication, soit tout de suite (dans le traitant de prise en compte de la communication), soit en différé (il y a alors un simple réveil de l'activité pour qui la communication est destinée, l'exécution de cette activité intervenant en fonction de l'auto-ordonnancement habituel). Une troisième possibilité consiste en un couplage de ces deux progressions par l'emploi simultané des deux solutions : une scrutation à faible coût est généralement faite, mais si les communications sont trop nombreuses vis à vis des scrutations, des interruptions sont alors générées (voir à ce propos [Maquelin et al. 1996]).

Nous avons présenté l'option prise par Athapascan-0a, qui consiste à profiter du tamponnage important réalisé par PVM pour ne scruter le réseau que lorsque c'est nécessaire. D'autres options sont possibles ; par exemple dans TPVM et Chant, la scrutation est réalisée tour à tour par chaque fil d'exécution qui souhaite recevoir un message. Cela conduit à un coût de gestion important mais permet une meilleure réactivité lors de l'utilisation du temps partagé. Dans PM2, un fil d'exécution accède au réseau en exclusion et sert d'intermédiaire vis à vis des autres. En particulier, il leur rend la main lorsque le réseau est bloquant. La progression des communications est donc aussi dépendante de l'ordonnancement des fils d'exécutions. Dans MPI-F, la progression des communications se fait à chaque action de l'utilisateur ; de plus une interruption périodique permet de forcer les communications lorsque de longs calculs sont effectués. A l'inverse, Panda génère une interruption lors de chaque arrivée de message ; ce sont donc les communications qui dirigent les calculs.

Les critères de l'interaction entre le noyau de communication et le noyau de multiprogrammation sont le coût de l'action de progression (en général faible pour une scrutation mais élevé pour une interruption), le temps de réponse (à l'inverse faible pour une interruption mais grand pour une scrutation, qui n'intervient que de temps en temps), et les fonctionnalités offertes dans un traitant de communication : possibilité de blocage, de création d'activités, de communication. Il est possible d'évaluer quantitativement ces différentes solutions, mais il nous semble difficile de choisir globalement LA

meilleure solution. En effet, nous pensons qu'elle dépend fortement de l'application et de la machine visées. Par exemple, le partage de temps et une forte réactivité aux communications sont importants pour les applications de recherche combinatoire ; par contre le surcoût d'exécution généré par ces mécanismes est généralement inutile pour les applications de calcul « pur » comme les simulations, ainsi que nous le montre l'application de dynamique moléculaire que nous avons détaillée.

L'exécutif résultant peut être caractérisé, comme nous avons essayé de le faire pour les divers exécutifs présentés, par :

- ses exigences sur le noyau de multiprogrammation (en général simplement création, suicide, identification, exclusion et ordonnancement explicite ; éventuellement préemption et priorités...),
- ses exigences sur le noyau de communication (en particulier en ce qui concerne la gestion des contextes, du stockage temporaire, mais aussi ses caractéristiques vis à vis de la multiprogrammation : sauf ou même synchrone vis à vis des fils d'exécution, et son API qui peut être standard ou bien exotique),
- son niveau qui peut être celui d'une bibliothèque de base (ressemblant à Posix + MPI) ou bien celui d'un « langage » présentant des concepts évolués (équipes, flots...),
- sa portabilité,
- son interopérabilité entre différents portages,
- son support de l'hétérogénéité de machines ou de réseaux,
- sa capacité d'exécution efficace sur une machines SMP (à parallélisme vrai).

Des critères d'efficacité peuvent aussi être cités, en particulier les coûts de base que sont la création d'un nouveau contexte, la création d'un processus léger, la commutation de contexte, la préemption (effet sur les caches...), la latence d'une communication, le débit des communications, la scrutation du réseau et enfin le coût d'une communication locale si la notion de mémoire partagée locale n'existe pas.

Si l'on devait classer ces exécutifs, nous retiendrions cinq axes :

- le modèle d'interaction distante selon trois possibilités : exécution à distance, échange de messages, accès mémoire distant,
- l'interaction locale : par mémoire partagée ou par l'un des mécanismes d'interaction distante,
- l'approche de conception : pragmatique (mise côte à côte), intégrée (développement couplé), idéaliste (basée sur une machine virtuelle),
- le niveau d'expression : semblable à « MPI+POSIX » ou introduisant des concepts de plus haut niveau (moniteurs...),
- le contrôle de l'exécution : ordonnancement des fils d'exécution (FIFO ou temps partagé), filtrage des messages (selon étiquette, émetteur, plan de communication, ...).

Notre but ici n'a pas été de trancher pour l'une ou l'autre des approches présentées, pour conclure que tel ou tel exécutif est le meilleur, mais d'introduire des points de comparaison permettant de débuter une réflexion globale sur l'étendue des possibilités et ce que pourrait être l'exécutif idéal.

Le lecteur intéressé trouvera un complément d'informations et plus de pointeurs sur l'intégration de la multiprogrammation légère et des communications à l'adresse :

http ://www-apache.imag.fr/threadedcomm.

Voir aussi l'article [Geib 1996] à propos de l'intérêt de la multiprogrammation légère pour la régulation de charge dans les applications parallèles.

Chapitre 6

Conclusion et perspectives

Nous concluons cette étude par un rappel de notre démarche pour un support exécutif pour applications irrégulières, l'exposé des choix principaux auxquels on est confronté lors de la réalisation d'un tel exécutif, une conclusion succincte de notre expérience sur Athapascan-0a et un énoncé des perspectives qui font suite à ce travail. Nous survolons en particulier Athapscan-0b, le successeur du prototype Athapascan-0a.

6.1 Parallélisme, machines et expression d'applications

En prélude de ce document, nous avons indiqué l'importance du parallélisme pour les années à venir ; à savoir la possibilité de traiter des problèmes de plus en plus grands plus vite. Diverses organisations de machines parallèles sont possibles ; les machines à mémoire distribuée offrant la plus grande puissance de crête actuellement. Dans la mesure où l'emploi de multiprocesseurs symétriques se généralise, nous pensons qu'à très court terme les machines parallèles suivront le modèle MMP que nous avons présenté, c'est à dire seront composées de plusieurs multiprocesseurs symétriques interconnectés par un réseau rapide à base de commutateurs. La programmation de telles machines accroîtra la nécessité de manipuler un grand nombre de séquences de calculs et de communications concurrentes. En effet, les multiprocesseurs symétriques prennent en charge plusieurs fils de calculs indifféremment ; les réseaux d'interconnexion à base de commutateurs prennent en charge plusieurs communications concurremment. La variabilité d'une machine à l'autre sera surtout définie par le grain de ces séquences à choisir en fonction des caractéristiques - de latence et de débit notamment - de la machine pour une exécution efficace de l'application.

Différentes formes d'expression d'application parallèles sont possibles. Certaines sont implicites et reportent l'effort de parallélisation sur le système de programmation. D'autres sont explicites, le programmeur devant lui-même exprimer les différentes séquences et leurs synchronisations. Au même titre qu'une bonne granularité, la localité d'accès aux données et l'équilibrage de charge sont primordiaux pour l'exécution rapide d'une application. Nous n'avons pas abordé ici les différentes façons de mettre à jour une « bonne » parallélisation d'une application ; nous nous sommes restreint à l'étude du support exécutif parallèle nécessaire quelque soit l'expression du parallélisme employée, et à ses deux concepts de base : les processus et les canaux de communications.

6.2 Un support exécutif portable pour applications parallèles irrégulières

Nous avons vu que les systèmes d'exploitation parallèles offrent un support pour le parallélisme adéquat ; cependant, ils ne sont généralement pas disponibles pour les nouvelles machines parallèles, qui possèdent un noyau de système classique sur chaque noeud. Les extensions de ces systèmes classiques, dédiées aux applications distribuées, ne conviennent pas pour les applications parallèles. La nécessité de définir un support d'exécution facilement portable amoindri l'intérêt des bibliothèques propriétaires pour le parallélisme au profit des bibliothèques portables comme PVM ou MPI. Mais celles-ci ne sont pas adaptées aux applications parallèles irrégulières car les processus manipulés sont lourds.

En effet, nous avons défini les applications parallèles irrégulières comme ayant un comportement imprévisible sauf à connaître les données de l'instance du problème à traiter. Pour ces applications, il est donc nécessaire de pouvoir exprimer et supporter l'exécution concurrente, dynamique, d'un grand nombre de séquences de grain éventuellement fin. En conséquence, une des caractéristiques principales d'un support exécutif parallèle pour applications irrégulières est le coût engendré par l'établissement d'une nouvelle séquence. On doit donc tout faire pour réduire ce coût [1] .

Au niveau des communications, l'emploi d'un protocole « léger » en espace utilisateur, comme celui de PVMe, est nécessaire (voir aussi [Henry & Joerg 1992, Bershad et al. 1991]). Au niveau des calculs, la prise en compte à faible coût d'un ensemble de séquences peut être réalisé de deux façons différentes :

- à l'aide d'un automate séquentiel, qui réagit à chaque communication entrante et fait progresser les calculs en conséquence. La poursuite de plusieurs séquences de calcul interrompues par la nécessité d'attendre de nouvelles communications peut être réalisée à l'aide de continuations.
- à l'aide de la multiprogrammation légère. Le mécanisme de la multiprogrammation effectue des changements de contexte automatiquement à chaque fois qu'une communication se révèle bloquante.

Si la méthode de l'automate à base de continuations est la plus performante (car elle ne sauvegarde que la « continuation », c'est à dire l'ensemble minimal de données nécessaire pour reprendre le calcul après interruption), elle est très complexe à prouver et à maintenir. Il est bien sûr possible de générer automatiquement un tel automate sous certaines conditions ; cependant cela nécessite l'emploi de techniques de compilation.

Notre choix dans cette étude a été de prendre l'approche plus simple, de multiprogrammation des communications, qui est applicable sans restrictions. Cette approche s'articule autour de deux fonctionnalités : création et synchronisation de fils d'exécution d'une part, communication de données d'autre part.

Nous pouvons distinguer cinq choix principaux dans la problèmatique d'un tel support d'exécution parallèle :

- Quelle interaction à distance ? Comment les différentes séquences de calculs vont elles interagir à distance ? Trois grandes solutions ont été montrées dans notre comparatif :
  - *L'exécution à distance.* L'interaction est limitée à la création de nouveaux fils d'exécution à distance ; en général la création est doublée d'une communication de données sous la forme de paramètres de création. C'est un mécanisme « tout-en-un » qui résout à la fois

1. Nous ne considérons pas ici les architectures de processeurs multiflots ([Iannucci et al. 1994, Agarwal et al. 1990, Alverson et al. 1990, Nikhil et al. 1992, Ang et al. 1994, Chiou et al. 1995, Sakai et al. 1993, Halstead, Jr. & Fujita 1988, Hum et al. 1994, Dally & al 1989, Spertus et al. 1993])

le problème de la création de nouvelles activités et la communication de données. La synchronisation entre activités concurrentes peut aussi être formalisée dans ce modèle (comme le RPC asynchrone employé par Athapascan-0a).

- *L'échange de messages.* L'échange de messages se fait entre activités existantes. Le surcoût d'utilisation d'un nouveau fil d'exécution à chaque communication est ainsi évité ; d'autre part ce modèle est plus souple pour les communications de données, en particulier de façon collective. Il est possible de réaliser des démons chargés de traiter des requêtes de synchronisation ou de création de nouvelles activités. Cependant, la structure du programme doit être prévue et le protocole de communication avec les démons établi.
- *L'accès mémoire à distance.* C'est une interaction qui ne nécessite pas de correspondant. Dans certaines implantations matérielles, le processeur cible n'est pas interrompu. Cette interaction permet aussi de réaliser des synchronisations à distance et de créer de nouveaux fils d'exécution, mais nécessite pour ce faire la conception d'un protocole particulier. L'implantation hétérogène peut être difficile car l'accès mémoire à distance repose naturellement sur une notion d'espace d'adressage homogène.

En fait ce sont trois facettes d'une même solution, qui devrait définir un protocole clair de création d'activité, de synchronisation et de communication de données. Ce protocole devrait accepter toutes les variantes possibles (avec ou sans création d'activité, avec ou sans synchronisation, avec ou sans activité correspondante), et pouvoir être implanté efficacement sur n'importe quelle architecture de machine. La notion de pointeur global est importante pour la mise en œuvre de ce protocole.

- Quelle interaction locale ? Dans le cadre de notre machine MMP, la communication entre activités situées sur le même nœud peut être réalisée par mémoire partagée. Une autre solution est d'employer une simplification de l'interaction employée à distance. Comme le montrent TPVM et LPVM, l'emploi ad-hoc de l'interaction employée à distance n'est guère satisfaisant. Notre opinion est qu'une interaction locale (par mémoire partagée) fondamentalement différente de celle employée à distance doit être mise en œuvre. Cela fait partie du contrôle de la localité ; nous en avons montré un exemple en Athapascan-0a lors de l'appel local du produit scalaire.
- Quelle base de réalisation ? Différentes bases sont possibles : PVM, MPI, les *sockets* pour n'en citer que les principales. Dans notre comparatif, nous avons distingué trois approches de réalisation :
  - *L'approche pragmatique.* Elle consiste à mettre côte à côte des noyaux préexistants, en les modifiant le moins possible. Elle présente l'avantage d'être simple à mettre en œuvre et d'être relativement portable.
  - *L'approche intégrée.* Elle consiste à reconstruire des communications en y intégrant la multiprogrammation légère. Il y a ainsi une meilleure adéquation, une meilleure efficacité. Par extension, nous appelons aussi intégrée l'approche qui consiste à employer pragmatiquement la base la plus appropriée en fonction de la machine cible, c'est à dire à offrir une seule interface mais plusieurs réalisations relativement différentes sur différentes machines.
  - *L'approche idéaliste.* Elle consiste à clairement définir une couche de portabilité. La réalisation est ainsi découpée en une couche portable et diverses implantations (intégrées ou pragmatiques) de la couche de portabilité. Cette approche permet d'assurer efficacement la portabilité, à condition que la couche de portabilité soit judicieusement définie.

Chacune de ces approches a ses avantages et ses inconvénients ; le choix doit être fait en fonction des buts de la réalisation.

- Quel niveau d'expression ? Les machines abstraites [Culler et al. 1991, Culler et al. 1993, Nikhil 1990, Hum et al. 1995] définissent un ensemble d'instructions de bas niveau, généralement comme cible d'un compilateur. Les bibliothèques d'opérateurs regroupent un ensemble

d'opérateurs permettant l'utilisation des fonctionnalités de l'exécutif depuis un langage séquentiel. La totalité des exécutifs que nous avons présentés rentrent dans la catégorie des bibliothèques d'opérateurs, puisque nous nous sommes principalement intéressé aux exécutifs programmables manuellement. La plupart de ces exécutifs présentent une interface du style « MPI+POSIX ». Certains rajoutent des notions que l'on ne trouve ni dans le noyau de communication, ni dans le noyau de multiprogrammation employés. C'est le cas par exemple pour les Ropes de Chant ou les moniteurs de DTh.

- Quel contrôle de l'exécution ? Différents contrôle d'exécution sont possibles. Les noyaux de multiprogrammation utilisent généralement un tourniquet FIFO et des mécanismes de priorités pour diriger l'exécution à l'aide d'un auto-ordonnanceur. Les noyaux de communication utilisent des filtres ou des gardes pour sélectionner les messages à traiter. Un ordonnancement global ou une politique de régulation de charge peuvent être implantés à l'aide des mécanismes d'information et de rétroaction fournis par l'exécutif. Nous avons vu d'autre part que l'asservissement des progressions des communications et des calculs n'est pas trivial.

Le but fondamental d'un support exécutif pour applications irrégulières est de permettre l'exécution de telles applications. Nous avons vu que la technique de multiprogrammation employée diminue drastiquement la durée de création d'un nouveau calcul, mais augmente le coût des communications. Ce surcoût peut être important si les communications natives sont fortement optimisées (MPI-F sur l'IBM SP1, par exemple). C'est particulièrement vrai pour la latence de communication. Cependant, ce surcoût est fixe et lorsque l'on utilise des messages de taille raisonnable, la perte d'efficacité est acceptable dans le cadre d'applications irrégulières. Les communications sont moins rapides mais le recouvrement des attentes de communication par des calculs permet en fait de gagner en vitesse d'exécution. La plupart des expériences menées montrent que ce genre d'exécutif est par contre peu adapté aux applications *régulières*. Comme l'ont remarqué certains auteurs, l'intérêt de ce genre d'exécutif est principalement de permettre une expression simple d'une application parallèle irrégulière, une meilleure efficacité pouvant être obtenue à l'aide d'un automate, mais au prix d'une plus grande complexité de mise en œuvre.

6.3 Le prototype Athapascan-0a

Nous avons réalisé Athapascan-0a dans le cadre du projet APACHE. Nous avons effectué certains choix de réalisation :

- le modèle d'interaction est le client-serveur (RPC) uniquement, pour valider l'approche poly-algorithmique,
- le contrôle de flux est répercuté sur l'application (ou la couche de régulation de charge),
- le filtrage proposé est minimal (seulement l'exclusion),
- des informations d'état et de charge, et des possibilités de rétroaction minimales sont fournies : modification des machines physique et virtuelle, placement d'activités. La migration n'est par contre pas possible,
- la mémoire peut être partagée localement, pour optimiser les interactions locales,
- les processus légers sont gérés de façon coopérative, pour simplifier le modèle vis à vis du programmeur d'application,
- le système d'exploitation présent sur chaque noeud est mis à profit pour réaliser les fonctionnalités annexes (fichiers, terminaux...),
- le niveau d'expression est celui d'un langage simple,
- la réalisation concerne principalement le couplage de PVM avec divers noyaux de multiprogrammation.

Ces choix impliquent certaines limitations, parmi lesquelles les plus importantes sont l'absence de prise en contre des architectures SMP et une certaine rigidité de l'auto- ordonnancement (pas de temps partagé ni de scrutation périodique).

Nous avons décrit l'implantation d'Athapascan-0a à l'aide d'un noyau de communication présentant l'API de PVM et de différents noyaux de multiprogrammation non préemptibles. Nous avons montré que cette implantation est simple. La multiprogrammation légère est accolée au noyau de communication sans modification de celui-ci. Cela autorise un très grande portabilité de l'exécutif, aussi bien vis à vis du noyau de communication que du noyau de multiprogrammation. Dans la mesure où le noyau de communication ne remonte pas d'interruption lors de l'arrivée d'un message, le cœur de notre exécutif est l'opération d'ordonnancement-scrutation du réseau qui permet de coupler la progression des calculs et celle des communications. Nous avons cité diverses réalisations possibles. Nous n'avons pas évalué ces diverses variations, dans la mesure où cette évaluation doit être faite en regard des applications visées. La méthode que nous avons implantée (scrutation sur actions de l'utilisateur ou quand aucun fil d'exécution n'est prêt) semble adaptée aux applications réalisées jusqu'à présent.

Nous avons montré que le surcoût engendré par la création (ou plus exactement la réutilisation) d'un fil d'exécution à chaque communication est raisonnable. Nous avons d'autre part monté que cette organisation permet un bon recouvrement des attentes de communication par des calculs utiles, même pour un petit nombre de fils d'exécution. Enfin, nous avons montré, par les applications réelles réalisées dans le cadre du projet APACHE, que le modèle proposé est adéquat dans le cadre des applications parallèles irrégulières.

L'implantation proposée présente bien évidemment certains défauts. Le premier est de ne proposer qu'un seul modèle d'interaction. Nous nous sommes volontairement limités dans le cadre de ce prototype, mais nous pensons qu'un tel exécutif devrait présenter les trois modèles d'interaction : client-serveur, échange de message, accès mémoire à distance. En particulier l'emploi de l'échange de messages permet d'éviter le coût de création d'activité et se révèle approprié dans les cas où l'application n'est pas complètement irrégulière mais possède des parties régulières. La notion de pointeur global, qui n'est pas présente dans le prototype, nous semble aussi être très intéressante lorsque l'on peut utiliser un espace mémoire global. Un autre défaut du prototype est de cacher PVM. Nous pensons qu'une meilleure approche aurait été de compléter PVM par des mécanismes client-serveur. La programmation d'applications mixtes, mélangeant régularité et irrégularité, serait ainsi possible. Notons que cette approche n'est guère plus difficile que celle que nous avons explorée. Enfin, le prototype proposé est probablement trop lié à PVM. Nous pensons maintenant qu'une meilleure approche est de définir une interface portable et adéquate, qui virtualise les mécanismes de communication nécessaires. En particulier, les défauts de PVM comme le trop grand nombre de copies des données, l'absence de contrôle de flux et de signalisation « légère » de l'arrivée d'un message, pourraient ainsi être éliminés.

L'intérêt de notre travail a donc été de montrer qu'un support exécutif basé sur l'adjonction d'un noyau de multiprogrammation à un noyau de communication comme PVM est non seulement simple à réalisé, mais efficace pour les applications parallèles irrégulières et en plus ne rend pas le modèle de programmation plus compliqué, il le simplifie même en permettant d'exprimer clairement les différentes séquences de calculs et leurs interactions, tout en limitant les problèmes de concurrence.

D'autre part, nous avons identifié le problème majeur de ce type d'exécutif, à savoir le couplage de la progression des calculs avec celle des communications. Nous avons illustré diverses approches possibles, aussi bien celle prise par notre prototype que celles évaluées dans le cadre d'autres projets de recherche.

Enfin, grâce à notre support d'exécution, nous avons pu aider l'illustration des concepts de polyalgorithmes et de décomposition procédurale parallèle, qui nous semblent fondamentaux pour la réa-

lisation d'applications parallèles efficacement portables.

Cet exécutif a servi de support pour la première période du projet APACHE et a permis aux autres axes comme la régulation de charge, la réexécution déterministe, la prise et la visualisation de traces, et enfin l'étude des applications parallèles, de démarrer. Nous n'avons pas présenté une évaluation poussée à l'aide d'applications réelles ; nous n'avons pas non plus abordé l'évaluation de mécanismes de contrôle comme la régulation dynamique de charge, dans la mesure où ces aspects sont étudiés par les autres axes du projet APACHE.

6.4 Les perspectives

Les perspectives de cette étude sont multiples et peuvent être résumées selon plusieurs grandes catégories :

La première perspective concerne l'optimisation du fonctionnement de l'exécutif. Comme nous l'avons vu il est nécessaire d'utiliser des communications efficaces et de réduire le coût de la multiprogrammation. Par exemple sur certaines machines PVM n'est pas très performant, principalement à cause de sa façon d'emballer les données dans des tampons ; l'utilisation de MPI peut améliorer les performances. MPI emballe les données différemment de PVM et nécessite un contrôle plus précis du tamponnage des communications ; en conséquence l'implantation que nous avons réalisée n'est pas portable directement et efficacement sur MPI. Nous y reviendrons dans la suite. D'autre part, nous n'avons pas évalué un forçage de la progression des communications par interruption périodique ; d'une façon plus générale, il reste à effectuer une étude comparative des différents modes de couplage calculs / communications, basée sur des applications réelles.

La seconde perspective concerne l'accroissement du contrôle de l'exécution. Le prototype que nous avons décrit n'offre qu'un contrôle minimal et complexe à mettre en œuvre : il s'agit de contrôler l'exécution des requêtes en synchronisant les fils d'exécution d'une tâche. Il est ainsi possible de forcer un ordonnancement particulier des fils. Différentes approches peuvent être employées pour obtenir un contrôle plus ou moins précis à un moindre coût de réalisation (c'est à dire de façon quasi-automatique) : au niveau le plus bas il s'agit de rendre modifiable l'ordonnanceur des processus légers et / ou d'utiliser une réception de messages par interruption ; d'une façon plus générale il s'agit de coupler les progressions des calculs et des communications, aussi bien au niveau local de la tâche qu'au niveau global du programme, comme par exemple en utilisant un ordonnancement par bandes. L'utilisation du partage de temps et de différents niveaux de priorités semble aussi importante, ainsi éventuellement que l'emploi d'un *yield* qui désigne le successeur (à la manière des coroutines).

La troisième perspective concerne l'évaluation de l'adéquation du prototype construit à différentes applications régulières ou irrégulières réelles. Comme nous l'avons dit, cela rentre dans le cadre de travail du projet APACHE et certains résultats sont dors et déjà disponibles en dynamique moléculaire par exemple.

La quatrième perspective concerne l'étude de la régulation de charge des multiples activités crées. Là encore le projet APACHE peut annoncer quelques résultats préliminaires, établis à l'aide de programmes synthétiques.

La cinquième perspective concerne l'interface et le paradigme de programmation à employer. Nous avons mis à jour l'importance de la **décomposition procédurale parallèle** pour ses qualités qui sont :

- le fait de décomposer un problème en sous-problèmes exécutables en parallèle,
- l'utilisation d'une interface procédurale permettant une expression simple de la décomposition et à l'inverse de la composition de calculs,

- la formalisation d'un protocole clair pour l'échange de données,
- et un schéma de désignation contextuelle locale.

A l'opposé l'utilisation de procédures barrières (dérivé du concept de *multiprocédure* [Puaut et al. 1991]) peut permettre une expression simple des synchronisations, tout en respectant une interface procédurale. Une **procédure barrière** est une procédure appelée par plusieurs activités, qui transmettent des arguments et se bloquent en attente des résultats. Lorsque tous les arguments sont présents (tous les appelants ont effectués l'appel), la procédure barrière est exécutée et renvoie ses résultats à chaque appelant. Le site d'exécution peut être quelconque, par exemple celui de l'un des appelants. La synchronisation peut être plus relâchée qu'une vraie barrière, si par exemple la procédure barrière commence son exécution dès que les premiers arguments sont reçus, et libère les appelants au fur et à mesure que les résultats sont calculés.

Dans ce modèle l'utilisation *de flots de communication* indépendants est importante puisqu'elle permet d'exprimer la concurrence des communications et de réaliser une progression *pipeline* des données. C'est l'un des inconvénient du modèle procédural : les paramètres sont habituellement transmis en bloc et aucun pipeline n'est possible entre les activités parallèles. L'introduction de flots de communication permettrait, après le démarrage d'une procédure, la récupération de nouveaux arguments, de façon pipeline entre l'appelant et l'appelé. En particulier, cela permettrait la transmission concurrente de tableaux, en utilisant un flot indépendant pour chaque tableau.

La notion « *d'équipe* » nous semble naturelle, aussi bien pour identifier les sous- procédures résultantes de la décomposition parallèle, que les « sur-procédures » qui se synchronisent lors de l'appel d'une procédure barrière. Cette notion d'équipe permet, comme pour les Ropes de Chant, une désignation contextuelle locale de chacun des membres. Des opérateurs de communication collective peuvent être définis à l'aide du groupe que représente l'équipe d'appelants ou d'appelés, ou de « frères » (tous ceux qui ont été appelés par un appel de procédures parallèles).

Le mécanisme de *pointeur global* nous semble être la base d'une notion plus générale « *d'entité accessible à distance* », qu'elle prenne un formalisme orienté-objet ou non. Il peut bien sûr être employé dans le cadre de l'accès mémoire à distance, mais d'une façon plus générale il est aussi utile pour accéder des entités dont la structure interne est masquée. C'est alors un paramètre d'une méthode d'accès à l'entité. Nous avons cité l'importance de pouvoir réaliser des accès à distance de façon à éviter des transmissions en cascades de paramètres tout le long de l'arbre d'appel.

La multiprogrammation légère, utilisée localement, nous semble être un atout pour rendre la programmation parallèle plus simple ; elle doit cependant être maîtrisée. Nous proposons un concept de « *cohorte* » - un ensemble de processus légers esclaves aidant un processus maître, par exemple pour gérer différents flots de communication. La cohorte pourrait être une simplification des opérateurs POSIX de création, terminaison, jointure et détachement de fils d'exécution locaux. Nous proposons aussi une notion de « *sac de communication* », qui permette de décrire de façon simple des séquences de calculs et de communications automatiquement entrelacées pour masquer les délais de communication, mais sans pour autant nécessiter l'utilisation de processus légers. Il s'agit ici de poser quelques contraintes d'exécution (pas de pile distincte pour chaque séquence) mais de gagner les temps d'allocation et de libération de fils d'exécution, ainsi que certains temps de sauvegarde de registres avec l'aide du compilateur. Ces notions sont décrites dans [Christaller et al. 1995*b*] et [Christaller et al. 1995*a*]

La plupart des notions présentées ici ne sont pas nouvelles ; c'est leur intégration dans un seul système de programmation qui serait novatrice et permettrait probablement une meilleure expression des applications irrégulières.

Ces perspectives sont actuellement en cours d'étude dans le cadre de la « seconde période » du projet APACHE. Nous allons maintenant survoler l'implantation d'Athapascan-0b, le successeur du prototype Athapascan-0a.

6.5 Athapascan-0b

Deux raisons principalement ont conduit à une redéfinition complète d'Athapascan-0a. La première est l'efficacité : sur la machine principale du laboratoire, l'IBM SP1, MPI-F est environ deux fois plus rapide que PVMe. La seconde raison tient au modèle d'Athapascan-0 que nous souhaitions faire évoluer : réalisation d'une bibliothèque pure, un plus grand contrôle sur l'exécution (expérimentation sur le temps partagé, les priorités), réalisation d'un mécanisme d'accès mémoire à distance, introduction des notions de procédure barrière, d'équipe, etc...

Actuellement, Athapascan-0b est en phase de prototypage. Il est réalisé au dessus de MPI- F/DCE *threads* ou MPI-CH/Solaris *threads*. Il est découpé en deux couches :

- la couche « aKernel » permet l'utilisation fil-sauve sérialisée de MPI et définit des sémaphores. MPI est employé en exclusion, les fils d'exécution ne sont pas visibles de l'extérieur, ils sont en compétition pour recevoir les messages de MPI. Un démon prend en charge les communications terminées et débloque les fils d'exécution bloqués en attente. L'exécution de ce démon se fait en temps partagé à une priorité quelconque (normalement la plus basse), mais elle peut aussi être forcée (primitive *akAdvance*).
- la couche Athapascan-0b proprement dite propose des primitives de synchronisation (sémaphores, verrous, variables de condition), la notion de « cohorte » (fils d'exécution locaux), les notions de RSR *threaded* et *non-threaded*, ainsi que le Send / Recv entre processus.

Ce prototype est encore en cours de stabilisation. Les notions d'équipe, de procédure barrière etc.. seront introduites comme une troisième couche. Ce travail sera certainement décrit en détail dans les futures thèses du projet APACHE.

Index

Bibliographie

Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. & Young, M. (1986), Mach : A new kernel foundation for unix development, *in* 'Proc. Summer 1986 USENIX Conf.', pp. 93–112.

Ackaouy, E. (1994), 'Multiplexing PVM for multi-threaded tasks', Graduate Class Paper.

Adamo, J.-M. (1996), ARCH, an object-oriented library for asynchronous and loosely synchronous system programming, Tech. Rep. CTC95-TR228, Cornell Theory Center, Ithaca, NY.

Agarwal, A., Lim, B.-H., Kranz, D. & Kubiatowicz, J. (1990), APRIL : a processor architecture for multiprocessing, *in* 'Proc. 17th Annual International Symposium on Computer Architecture', IEEE Computer society and ACM SIGARCH, Seatle, Washington, pp. 104–114. Computer Architecture News, 18(2), Jun. 1990.

Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A. & Smith, B. (1990), The Tera computer system, *in* 'Proc. International Conference on Supercomputing', ACM, Amsterdam, The Netherlands, pp. 1–6. Computer Arhcitecture News, 18(3), Sep. 1990.

Anderson, T. E. (1990), *FastThreads user's manual*, DCSE, Univ. Washington, WA. in the Quartz distribution, ftp ://ftp.cs.washington/pub/Quartz1.0.tar.Z.

Anderson, T. E., Bershad, B. N., Lazowska, E. D. & Levy, H. M. (1992), 'Scheduler activations : efficient kernel support for the user-level management of parallelism', *ACM Trans. on Computer Systems* **10**, 53–79. also Proc. of the Thirteenth ACM Symposium on Operating Systems Principles, oct. 1991, pp. 95-105.

Andrews, G. R. & Schneider, F. B. (1983), 'Concepts and notations for concurrent programming', *Computing Surveys* **15**(1), 3–44.

Ang, B. S., Arvind & Chiou, D. (1994), StarT the next generation : integrating global caches and dataflow architectures, CSG Memo 354, CSG, LCS, MIT, Cambridge, MA.

Arbab, F. (1994*a*), *Build : A utility for building distributed and parallel applications*, Departement of interactive systems, CWI, Amsterdam.

Arbab, F. (1994*b*), *Config : A utility for run-time configuration of distributed applications*, Departement of interactive systems, CWI, Amsterdam.

Arbab, F. (1994*c*), Dth : A package for building applications with distributed threads, Tech. rep., Centrum voor Wiskunde en Informatica.

Arbab, F., Herman, I. & Spilling, P. (1993), 'An overview of Manifold and its implementation', *Concurrency : Practice and Experience* **5**(1), 23–71.

Armand, F., Herrman, F., Kipkis, J. & Rozier, M. (1990), Multi-threaded processes in CHORUS/MIX, *in* 'Proc. EEUG Spring'90 Conf.', pp. 1–13.

Arrouye, Y. (1995), Environnements de visualisation pour l'évaluation des performances des systèmes parallèles, PhD thesis, Institut National Polytechnique de Grenoble, France.

Bal, H. E., Kaashoek, M. F. & Tanenbaum, A. S. (1992), 'Orca : A language for parallel programming of distributed systems', *IEEE Transactions on Software Engineering* **18**(3), 190–205.

Bal, H. E., Steiner, J. G. & Tanenbaum, A. S. (1989), 'Programming languages for distributed computing systems', *ACM Computing Surveys* **21**(3), 261–322.

Bala, V., Kipnis, S., Rudolph, L. & Snir, M. (1992), Design efficient, scalable, and portable collective communication libraries, Tech. report preprint, IBM T.J.Watson Reseach Center.

Barnes, J. (1991), *Programmer en ADA*, InterEditions. ("Programming in Ada", Addison-Wesley).

Barnett, M., Gupta, S., Payne, D. P., Shuler, L., van de Geijn, R. & Watts, J. (1994), Interprocessor collective communication library (intercom), *in* 'Proc. of the Scalable High-Performance Computing Conference (SHPCC)', IEEE, pp. 357–364.

Barton-Davis, P., McNamee, D., Vaswani, R. & Lazowska, E. D. (1992), Ading scheduler activations to Mach 3.0, Tech. Rep. 92-08-03, DCSE, Univ. of Washington.

B.Carter, J., K.Bennett, J. & Zwaenepoel, W. (1991), Implementation and performance of munin, *in* 'Proc. 13th ACM Symp. on Oper. Syst. Princ.', pp. 152–164.

Bernard, P. & Trystram, D. (1996), Algorithme Parallèle de Dynamique Moléculaire, Rapport APACHE 20, LMC-IMAG, Grenoble, France.

Bernard, P.-E., Trystram, D. & Chapron, Y. (1996*a*), Parallélisation d'algorithmes de dynamique moléculaire, *in* 'Huitièmes Rencontres du Parallélisme', Bordeaux.

Bernard, P., Plateau, B. & Trystram, D. (1996*b*), Using threads for developing applications : Molecular dynamics as a case study, *in* 'Parallel Numerics 96', Gozd Martuljek, Slovenia.

Bernaschi, M. & Richelli, G. (1995), 'Development and results of PVMe on the IBM 9076 SP1', *Journal of Parallel and Distributed Computing* **29**, 75–83.

Bershad, B. N., Anderson, T. E., Lazowska, E. D. & Levy, H. M. (1990), 'Lightweight remote procedure call', *ACM Trans. on Computer Systems* **8**(1), 37–55.

Bershad, B. N., Anderson, T. E., Lazowska, E. D. & Levy, H. M. (1991), 'User-level interprocess communication for shared memory multiprocessors', *ACM Trans. on Comp. Sys.* **9**(2), 175–198.

Bershad, B. N., Lazowska, E. D. & Levy, H. M. (1988), 'PRESTO : A system for object-oriented parallel programming', *Software, Pratice and Experience* **18**(8), 713–732.

B.Herrmann & L.Philippe (1992), CHORUS/MIX, a distributed UNIX on multicomputers, Tech. Rep. CS/TR-92-10, Chorus systèmes, France.

Bhoedjang, R., Rumlhl, T., R. Hofman a, d. K. L. & Bal, H. (1993), Panda : A portable platform to support parallel programming languages, *in* 'Proc. Symp. on Experiences with Distributed and Multiprocessor Systems IV', pp. 213–226.

Birrell, A. & Nelson, B. (1984), 'Implementing remote procedure calls', *ACM Trans. on Comp. Sys.* **2**, 39–59.

Birrell, A. D. (1989), An introduction to programming with threads, Tech. Rep. 35, Digital Equipments Co. also in *Systems Programming with Modula-3*, Greg Nelson ed., Prentice Hall, 1991.

Bloomer, J. (1992), *Power programming with RPC*, O'Reilly.

Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K. & Zhou, Y. (1995), Cilk : An efficient multithreaded runtime system, *in* 'Proc. Symp. on Princ. and Prac. of Parallel Prog.', ACM, pp. 207–216.

Boothe, R. & Ranade, A. (1992), 'Improved multithreading techniques for hiding communication latency in multiprocessors', *ACM SIGARCH Computer Architecture News* **20**(2), 214–223.

Branstetter, M. L., Guse, J. A. & Nesset, D. M. (1991), ELROS : An Embedded Language for Remote Operations Service, Tech. Report UCRL-JC-108862, Lawrence Livermore National Laboratory.

Briat, J., Carissimi, A. S., Kannat, S. & Morel, E. (1996), Task scheduling for parallel execution of logic programs on distributed memory architectures, *in* 'TDP'96 : International Conference on Telecommunication, Distribution and Parallelism', Agelonde, La Londe Les Maures - France.

Briat, J., Christaller, M. & Roch, J.-L. (1994), Une maquette pour ATHAPASCAN-0, *in* L. Bougé, ed., 'Actes des 6èmes Rencontres Francophones du Parallélisme, RenPar'6', Ecole Normale Supérieure de Lyon, Ecole Nationale Supérieure, Lyon, France, pp. 231–235.

Briat, J., de Kergommeaux, J. C., Plateau, B., Roch, J.-L., Trystram, D., Villard, G. & Vincent, J.-M. (1993), APACHE : Algorithmique Parallèle et pArtage de CHargE, *in* 'Actes des 5èmes Rencontres Francophones du Parallélisme, RenPar'5', Brest, France.

Briat, J., Kannat, S. & Morel, E. (1995*a*), An environment to study dynamic load balancing functions and its application to the parallel logic system PLoSys, *in* 'CompuLog-Net'95 : WorkShop on Parallelism and Implementation Technologies', Utrecht - Netherlands.

Briat, J., Kannat, S. & Morel, E. (1995*b*), Plate-forme d'evaluation de strategies de regulation dynamique de charge pour le systeme logique parallele PLoSys, *in* 'RenPar'7 : 7eme Rencontre sur le Parallelisme', Faculte Polytechnique de Mons - Belgique.

Buhr, P. & Stroobosscher, R. (1990), 'The μ system : Providing light-weight concurrency on shared-memory multiprocessor systems running Unix', *Software Practice and Experience* **20**(9), 929–964.

Buhr, P. A., Ditchfield, G., Strobosscher, R. A. & Younger, B. M. (1992), 'μC++ : Concurrency in the object oriented language C++', *Software Practice and Experience* **22**(2), 137–172.

Bui, A. (1994), Etude analytique d'algorithmes distribués de routage, PhD thesis, Paris 7.

Burns, G., Daoud, R. & Vaigl, J. (1994), 'LAM : An open cluster environment for MPI'. Available by ftp from tbag.osc.edu in pub/lam.

Butler, R. & Lusk, E. (1992), User's guide to the p4 parallel programming system, Tech. Report ANL-92-17, Argonne National Laboratory.

Butler, R. & Lusk, E. (1994), 'Monitors, messages, and clusters : The p4 parallel programming system', *Parallel Computing* **20**, 547–564. also MCS, Argonne National Laboratory TR. MCS-P362-0493.

Cabrera-Dantart, R., Demeure, I. & Meunier, P. (1994), Phosphorus : Adding sharing memory to PVM, *in* 'Proc. First European PVM User's Group Meeting', IBM-ECSEC, ENS-Lyon, CASPUR-Università di Roma, Roma, Italy.

Carriero, N. & Gelernter, D. (1989), 'LINDA in context', *Communications of the ACM* **32**(4), 444–458.

Carriero, N. & Gelernter, D. (1990), *How to write parallel programs*, MIT Press.

Castaneda-Retiz, M.-R. & Plateau, B. (1996), Evaluation des stratégies de régulation de charge dynamique, *in* 'Huitièmes Rencontres du Parallélisme', Bordeaux.

Chandy, K. M. & Kesselman, C. (1993*a*), CC++ : A declarative concurrent object oriented programming notation, *in* 'Research Directions in Object Oriented Programming', MIT Press. also CalTech Tech.Rep. CS-TR-92-01.

Chandy, K. M. & Kesselman, C. (1993*b*), Compositional C++ : Compositional parallel programming, *in* Springer-Verlag, ed., 'Proc. Fifth Int'l Workshop on Parallel Languages and Compilers'.

Chapman, B., Haines, M., Mehrotra, P., Zima, H. & Rosendale, J. V. (1995), Opus : A coordination language for multidisciplinary applications, Technical report, ICASE, NASA LRC.

Chapman, B., Mehrotra, P., Rosendale, J. V. & Zima, H. (1994), A software architecture for multidisciplinary applications : integrating task and data parallelism, *in* 'Proc. CONPAR'94-VAPP VI 3rd Intl. Conf. on Vector and Parallel Processing, LNCS 854', ICASE, NASA LRC, Springer Verlag, Linz, Austria, pp. 664–676. also ICASE, NASA LRC, TR 94-54.

Cheriton, D. (1988), VMTP : Versatile Message Transaction Protocol, Technical Report RFC 1045, Stanford University.

Chiou, D., Ang, B. S., Greiner, R., Arvind, Hoe, J. C., Berckle, M. J., Hicks, J. E. & Boughton, A. (1995), StarT-NG : Delivering seamless parallel computing, *in* 'Proc. First Intl. EUROPAR Conf.', Springer-Verlag, Stockholm, Sweden, pp. 101–116. LNCS 966.

Chowdappa, A. K., Skjellum, A. & Doss, N. E. (1994), Thread-safe message passing with P4 and MPI, Tech. Report TR-CS-941025, Mississippi State University - Dept. of Computer Science.

Christaller, M. (1994*a*), ATHAPASCAN-0A : A control parallelism approach on top of PVM, *in* 'Proc. 1994 PVM User's Group Meeting', Garden Plaza Hotel, Oak Ridge, Tennessee.

Christaller, M. (1994*b*), ATHAPASCAN-0A sur PVM 3 : définition et mode d'emploi, Tech. Report Apache TR-11, IMAG Institute - APACHE team, Grenoble, France.

Christaller, M. (1994*c*), ATHAPASCAN-0B for PVM : adding threads to PVM, *in* 'Proc. First European PVM User's Group Meeting', IBM-ECSEC, ENS-Lyon, CASPUR-Università di Roma, Roma, Italy.

Christaller, M. & Castaneda-Retiz, M.-R. (1996), 'Parallélisme de contrôle sur PVM : l'expérience ATHAPASCAN', *Calculateurs Parallèles* **8**(2), 183–187.

Christaller, M., Briat, J. & Rivière, M. (1995*a*), 'ATHAPASCAN-0 : concepts structurants simples pour une programmation parallèle efficace', *Calculateurs Parallèles* **7**(2), 173–196.

Christaller, M., Briat, J. & Rivière, M. (1995*b*), ATHAPASCAN-0 : vers une expression du parallélisme à l'aide de décompositions en procédures parallèles et procédures barrières, *in* G. Libert, J.-L. Dekeyser & P. Manneback, eds, 'Actes des 7èmes Rencontres Francophones du Parallélisme, RenPar'7', P.I.P., Faculté Polytechnique de Mons, Belgique, pp. 217–220.

Christaller, M., Castaneda-Retiz, M.-R. & Gautier, T. (1995*c*), Control parallelism on top of PVM : The ATHAPASCAN environment, *in* J. Dongarra, M. Gengler, B. Tourancheau & X. Vigouroux, eds, 'Proc. Second European PVM User's Group Meeting', Hermes, Ecole Nationale Supérieure, Lyon, France, pp. 71–76.

Chuang, W. (1994), PVM light weight process package, Computation Structures Group Memo 372, LCS, MIT.

Coffin, M. & Olsson, R. A. (1989), 'An SR approach to multiway rendezvous', *Comput. Lang.* **14**(4), 255–262.

Colin, J.-N. (1995*a*), DTMS : A framework for multigrain distributed computing, *in* 'Proc. Hinet'95', IEEE Computer Society Press, pp. 83–89.

Colin, J.-N. (1995*b*), DTMS : Un environnement pour la programmation distribuée à grain déterminé, PhD thesis, Université de Mons-Hainaut.

Colin, J.-N., Geib, J.-M. & Libert, G. (1995), Using threads to implement parallel tasks, *in* 'Proc. ParCo'95'.

Comer, D. E. & Stevens, D. L. (1993), *Internetworking with TCP-IP : 3 : client-server programming and applications (BSD socket version).*

Cooper, E. C. & Draves, R. P. (1988), C threads, Tech Report CMU-CS-88-154, Carnegie Mellon University, School of Computer Science, Pittsburg, PA.

Crane, S. (1993), The REX lightweight process library, Conputer science technical report, Imperial College of Science and Technology, London, England. available at gummo.doc.ic.ac.uk.

Cruw-Neira, C., Sandin, D. J. & DeFanti, T. A. (1993), Surround-screen projection-based virtual reality : the design and implementation of the CAVE, *in* 'Computer graphics (Proc. SIGGRAPH'93)', ACM SIGGRAPH, pp. 135–142.

Culler, D. E., Goldstein, S. C., Schauser, K. E. & von Eicken, T. (1993), 'TAM - a compiler controlled threaded abstract machine', *J. of Para. and Distrib. Comp.* **18**, 347–370.

Culler, D. E., Sah, A., Schauser, K. E., von Eicken, T. & Wawrzynek, J. (1991), Fine-grain parallelism with minimal hardware support : A compiler-controlled threaded abstract machine, *in* '4th International Conference on Architectural Support for Programming Languages and Operating Systems', ACM SIGARCH, SIGPLAN, SIGOPS and the IEEE Computer Society, Santa Clara, Ca. Computer Architecture News, 19(2), Apr. 1991 ; Operating Systems Review, 25, Apr. 1991 ; SIGPLAN Notices, 26(4), Apr. 1991.

Dally, W. J. & al (1989), 'The J-Machine : A fine-grain concurrent computer', *Information Processing 89.*

de Kergommeaux, J. C. & Fagot, A. (1995), Environnement d'aide à la mise au point de programmes parallèles, *in* 'Actes des 7-ièmes Rencontres francophones du Parallélisme, RenPar'7', Faculté Polytechnique de Mons (Belgique).

Dean, R. (1993), Using continuations to build a user-level threads library, *in* 'Proc. of the third USENIX Mach Conf.', pp. 136–151.

Deering, S. E. & Cheriton, D. R. (91), 'Multicast routing in datagram internetworks and extended LANs', *ACM Trans. on Computer Systems*.

Demeure, I. & Farhat, J. (1994), 'Systèmes de processus légers : concepts et exemples', *Tech. et Sc. Infor.* **13**(6), 765–795.

Desprez, F. & Tourancheau, B. (1994), 'LOCCS : Low overhead communication and computation subroutines', *Future Generation Computer Systems* pp. 279–284.

Digital Equipment Corp. (1992), *DECthreads : Guide to DECthreads*, Digital Equipment Corp.

Dijkstra, E. W. (1965), *Programming Languages*, Academic Press, chapter Cooperating Sequential Processes.

Doeppner Jr., T. W. (1987), Threads : a system for the support of concurrent programming, Tech. Report TR CS-87-11, Brown University, Dept. of Computer Science.

Doeppner, T. (1993), The thread-monitor library : A system for monitoring Solaris-threads programs, Tech. Rep. CS-93-53, DCS, Brown University.

Dongarra, J., Geist, A., Manchek, R. & Sunderam, V. (1993), 'Integrated PVM framework supports heterogeneous network computing', *Computers in Physics* **7**(2), 166–175.

Doss, N., Gropp, W., Lusk, E. & Skjellum, A. (1993), An initial implementation of MPI, Tech. Report MPCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL.

Draves, R. P., Bershad, B. N., Rashid, R. F. & Dean, R. W. (1991), Using continuations to implement thread management and communication in operating systems, *in* 'Proc. 13th ACM Symp. on Operating Systems'.

Edinburgh Parallel Computing Centre (1991), *CHIMP Concepts*, University of Edinburgh.

Edinburgh Parallel Computing Centre (1992), *CHIMP version 1.0 Interface*, University of Edinburgh.

Elmasri, N., Hum, H. H. J. & Gao, G. R. (1994), The Threaded Communication Library : Preliminary experiences on a multiprocessor with dual processor nodes, ACAPS Tech. Memo 89, SGS, McGill Univ., Montréal, Québec.

Engler, D. R., Andrews, G. R. & Lowenthal, D. K. (1993), Filaments : Efficient support for fine-grain parallelism, Tech. Report TR 93-13, University of Arizona, Tucson, Arizona.

Eykholt, J. R., Kleiman, S. R., Barton, S., Faulkner, R., Shivalingiah, A., Smith, M., Stein, D., Voll, J., Weeks, M. & Williams, D. (1992), Beyond multiprocessing - multithreading the SunOS kernel, *in* 'Proc. of the Usenix Conference', SunSoft Inc., pp. 11–18.

Fagot, A. & de Kergommeaux, J. C. (1994), Optimized record-replay mechanism for RPC-based parallel programming, *in* B. Verlag, ed., 'Proc. of IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems', Ascona, Switzerland, pp. 347–352.

Fagot, A. & de Kergommeaux, J. C. (1995*a*), Formal and experimental validation of a low-overhead execution replay mechanism, *in* S. Haridi, K. Ali & P. Magnusson, eds, 'Euro-Par'95 Parallel Processing', Vol. 966 of *LNCS*, Springer-Verlag, pp. 167–178.

Fagot, A. & de Kergommeaux, J. C. (1995*b*), Optimized execution replay mechanism for RPC-based parallel programming models, Rapport APACHE 18, LMC-IMAG, Grenoble, France. Disponible en ftp anonyme : `ftp.imag.fr:imag/APACHE/RAPPORTS`.

Fagot, A. & de Kergommeaux, J. C. (1996), Systematic assessment of the overhead of tracing parallel programs, *in* E. Zapata, ed., 'Proceedings of the 4th Euromicro Workshop on Parallel and Distributed processing, PDP'96', IEEE/CS, Braga.

Fahringer, T., Haines, M. & Mehrotra, P. (1995), On the utility of threads for data parallel programming, *in* 'Proc. of the ACM Intl Conf. on Supercomputing', Barcelona, Spain. also ICASE, NASA LRC Tech.Rep. 95-35.

Felton, E. & McNamee, D. (1992*a*), *NewThreads 2.0 User's Guide*.

Felton, E. W. & McNamee, D. (1992*b*), Improving the performance of message passing applications by multithreading, *in* 'Proc. of the Scalable High Performance Computing Conference', pp. 84–89.

Ferrari, A. & Sunderam, V. (1995*a*), TPVM : Distributed concurrent computing with lightweight processes, *in* 'Proc. of IEEE High Performance Distributed Computing', Washington, D. C., pp. 211–218.

Ferrari, A. & Sunderam, V. S. (1994), TPVM : A threads-based interface and subsystem for PVM, Tech. Rep. CSTR-940802, Univ. Virginia.

Ferrari, A. & Sunderam, V. S. (1995*b*), Multiparadigm distributed computing with pvm, Technical Report CSTR-951201, Dept. Math. and Comp. Sc., Emory Univ., Atlanta, USA.

Flower, J., Kolawa, A. & Bharadwaj, S. (1991), 'The Express way to distributed processing', *Supercomputing Review* pp. 54–55.

Ford, B. & Lepreau, J. (1993), Evolving mach 3.0 to a migrating thread model, Technical Report UUCS-93-022, Univ. Utah.

Foster, I. (1995), *Designing and Building Parallel Programs*, Addison-Wesley.

Foster, I. & Chandy, K. M. (1994), 'Fortran M : A language for modular parallel programming', *J. Parallel and Distributed Computing*. also ANL Tech.Rep. MCS-P327-0992.

Foster, I. & Olson, R. (1995), A guide to parallel and distributed programming in nPerl, Technical report, MCS, Argonne National Lab, Argonne, IL.

Foster, I., Garnett, J. & Tuecke, S. (1994*a*), *Nexus User's Guide*. version 2.0.

Foster, I., Geisler, J., Kesselman, C. & Tuecke, S. (1996*a*), Multimethod communication for high-performance networked computing systems, Technical report, MCS, Argonne National Laboratory.

Foster, I., Geisler, J., Nickless, W. & Tuecke, S. (1996*b*), Software infrastructure for high-performance distributed computing, Preprint, MCS, Argonne National Lab, Argonne, IL.

Foster, I., Kesselman, C. & Snir, M. (1995), Generalized communicators in mpi, draft.

Foster, I., Kesselman, C. & Tuecke, S. (1994*b*), Nexus : Runtime support for task-parallel programming languages, Technical report, MCS, ANL.

Foster, I., Kesselman, C. & Tuecke, S. (1994*c*), The Nexus task-parallel runtime system, *in* 'Proc. 1st Workshop on Parallel Processing', Tata McGraw Hill, pp. 457–462.

Foster, I., Kesselman, C. & Tuecke, S. (1994*d*), Portable mechanisms for multithreaded distributed computations, Technical report, MCS, ANL.

Foster, I., Kesselman, C. & Tuecke, S. (1996*c*), The Nexus approach to integrating multithreading and communication, *in* D. T. J. Chassin de Kergommeaux, ed., 'Parallel Programming Environments for High Performance Computing, ESPPE'96', 2nd European School of Computer Science, Alpes d'Huerz, France, pp. 53–67.

Foster, I., Kesselman, C., Olson, R. & Tuecke, S. (1994*e*), Nexus : An interoperability toolkit for parallel and distributed computer systems, Technical Report ANL/MCS-TM-189, MCS, Argonne National Lab, Argonne, IL.

Foster, I., Kesselman, C., Schwab, S. & Tuecke, S. (1996*d*), Configuration of complex distributed and parallel computations, Technical report, MCS, ANL, Argonne, IL.

Foster, I., Olson, B. & Tuecke, S. (1993), Programming in Fortran M, Tech. Report ANL-93/26, Argonne National Laboratoty.

Franke, H. (1994*a*), *MPI-F : An MPI Implementation for IBM SP1/SP2*, IBM T.J.Watson Research Center, Yorktown Heights, NY.

Franke, H. (1994*b*), Multi-threading under MPI, Technical report, IBM T.J.Watson Research Center, Yorktown Heights, NY.

Franke, H., Hochschild, P., Pattnaik, P. & Snir, M. (1994), An efficient implementation of MPI on IBM-SP1, *in* 'Proc. of the 1994 International Conference on Parallel Processing'.

Freeh, V. W., Loventhal, D. K. & Andrews, G. R. (1994*a*), Distributed filaments : Fine-grain parallelism on a cluster of workstations, *in* 'Proc. First Symposium on Operating Systems Design and Implementation', Usenix Association, pp. 201–213.

Freeh, V. W., Lowenthal, D. K. & Andrews, G. R. (1994*b*), Distributed filaments : Efficient fine-grain parallelism on a cluster of workstations, *in* 'Proc. 1st Symp. On Operating Systems Design and Implementation', Usenix Association, pp. 201–213.

Gautier, T. (1996), Calcul formel et parallélisme : Conception du Système Givaro et Applications au Calcul dans les Extensions Algébriques, PhD thesis, INP Grenoble.

Gautier, T. & Roch, J. (1994), PAC++ System and Parallel Algebraic Numbers Computation, *in* H. Hong, ed., 'First International Symposium on Parallel Symbolic Computation (PASCO'94)', Vol. 5 of *Lecture Notes Series in Computing*, p. 145.

Gautier, T., Roch, J. & Villard, G. (1995), Regular versus irregular problems and algorithms, *in* 'Proc. of IRREGULAR'95, Lyon, France', Springer-Verlag.

Gautier, T., Roch, J.-L. & Villard, G. (1994), PAC++ v2.0 User and developer manual, Rapport APACHE 14, Rapport APACHE.

Gehani, N. H. & Roome, W. D. (1986), 'Concurrent C', *Software, Practice and Experience* **16**(9), 821–844.

Geib, J.-M. (1996), Processus légers distribués et régulation de charge, *in* 'Proc. Ecole Placement Dynamique et Régulation de Charge, Giens, France', pp. 89–102.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. & Sunderam, V. (1993), PVM 3 users's guide and reference manual, Tech. Report ORNL/TM-12187, Oak Ridge National Laboratory.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. & Sunderam, V. (1994), *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*, The MIT Press, Cambridge, Massachusetts. Electronically available at ftp://netlib2.cs.utk.edu/pvm3/book/pvm-book.ps, http://www.netlib.org/pvm3/book/pvm-book.html.

Geist, G. A., Heath, M. T., Peyton, B. W. & Worley, P. H. (1992), A user's guide to PICL : a portable instrumented communication library, Tech. Report TM-11616, Oak Ridge National Laboratory.

Gibbons, P. B. (1987), 'A stub generator for multi-language RPC in heterogeneous environments', *IEEE Transactions on Software Engineering* **13**(1), 77–87.

Giering, E. W. & Baker, T. P. (1992), Using POSIX threads to implement Ada tasking : description of work in progress, *in* 'Proc. ACM TriAda Conf.'.

Goldstein, S., Schauser, K. E. & Culler, D. (1995), Lazy threads, stacklets, and synchronizers : enabling primitives for compiling parallel languages, *in* 'Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers', Troy, NY.

Goossens, B. & Vu, D. T. (1996), On-chip multiprocessing, *in* L. Bougé, P. Fraigniaud, A. Mignotte & Y. Robert, eds, 'Proc. EuroPar'96 (Parallel Processing), Lyon, France, LNCS 1124', pp. 789–796.

Grant, B., Skjellum, A. & Burton, L. (1992), The PVM systems : An in-depth analysis and documenting study - concise edition, Tech. Report UCRL-JC-112016, Lawrence Livermore National Laboratory.

Gropp, W. D. & Smith, B. (1993), Chameleon parallel programming tools user's manual, Tech. Report ANL-93-23, Argonne National Laboratory, Argonne, IL.

Gropp, W., Lusk, E. & Skjellum, A. (1995), *Using MPI : Portable Parallel Programming with the Message Passing Interface*, MIT Press.

Gropp, W., Lusk, E., Doss, N. & Skjellum, A. (1996), A high-performance, portable implementation of the mpi message passing interface standard, Preprint MCS-P567-0296, MCS, Argonne National Lab, Argonne, IL.

Hagersten, E., Landin, A. & Haridi, S. (1992), 'DDM - a cache-only memory architecture', *IEEE Computer* **25**(9), 4–54.

Haines, M. & Böhm, W. (1993*a*), An evaluation of software multithreading in a conventional distributed memory multiprocessor, *in* 'IEEE Symposium on Parallel and Distributed Processing', IEEE, pp. 106–113.

Haines, M. & Böhm, W. (1993*b*), An initial comparison of implicit and explicit programming styles for distributed memory multiprocessors, Technical report, ICASE, NASA LRC.

Haines, M., Cronk, D. & Mehrotra, P. (1994*a*), *The Chant user's guide*, ICASE, NASA LRC, Hampton, VA.

Haines, M., Cronk, D. & Mehrotra, P. (1994*b*), On the design of Chant: A talking threads package, *in* 'Proc. Supercomputing'94', Washington, D.C., pp. 350–359. also ICASE Tech.Rep. 94-25.

Haines, M., Hess, B., Mehrotra, P., Rosendale, J. V. & Zima, H. (1995*a*), Rutime support for data parallel tasks, *in* 'Proc. 5th Symp. on the Frontiers of Massively Parallel Computation', McLean, VA, pp. 432–439. also ICASE, NASA LRC Tech.Rep. TR 94-26.

Haines, M., Mehrotra, P. & Cronk, D. (1994*c*), Chant: lightweight threads in a distributed memory environment, Technical report, ICASE, NASA LRC.

Haines, M., Mehrotra, P. & Cronk, D. (1995*b*), Ropes: Support for collective operations among distributed threads, TR 95-36, ICASE, NASA LRC, Hampton, VA.

Halstead, Jr., R. H. & Fujita, T. (1988), MASA: A multithreaded processor architecture for parallel symbolic computing, *in* 'Proc. 15th Annual International Symp. on Computer Architecture', Honolulu, Hawaii, pp. 443–451.

HelmBold, D. P. & McDowell, C. E. (1994), A taxonomy of race conditions, Technical Report UCSC-CRL-94-34, Univ. of California, Santa Cruz.

Henry, D. S. & Joerg, C. F. (1992), A tightly coupled processor-network interface, *in* 'Proc. Fifth Intl. Conf. on Architectural Support for Programming Langages and Operating Systems', Boston, Ma, pp. 111–122.

Hinton, J. & Pinder, A. (1993), *Transputer Hardware and System Design*, Prentice Hall.

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

Hum, H. H. J., Maquelin, O., Theobald, K. B., Tian, X., Tang, X., Gao, G. R., Cupryk, P., Elmasri, N., Hendren, L. J., Jimenez, A., Krishnan, S., Marquez, A., Merali, S., Nemawarkar, S., Panangaden, P., Xue, X. & Zhu, Y. (1994), The multi-threaded architecture multiprocessor, ACAPS Tech. Memo 88, SGS, McGill Univ., Montréal, Québec.

Hum, H. H. J., Maquelin, O., Theobald, K. B., Tian, X., Tang, X., Gao, G. R., Cupryk, P., Elmasri, N., Hendren, L. J., Jimenez, A., Krishnan, S., Marquez, A., Merali, S., Nemawarkar, S., Panangaden, P., Xue, X. & Zhu, Y. (1995), A design study of the EARTH multiprocessor, *in* 'Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'95)', Limassol, Cyprus, pp. 59–68.

Iannucci, R. A., Gao, G. R., Halstead Jr., R. H. & Smith, B., eds (1994), *Multithreaded Computer Architecture : A summary of the state of the Art*, Kluwer Academic Publishers, Norwell, Ma.

IBM (1994), *IBM AIX PVMe User's Guide and Subroutine Reference*, release 2.0 edn, IBM.

Inohara, S. & Masuda, T. (1994), A framework for minimizing thread management overhead based on asynchronous cooperation between user and kernel schedulers, Technical Report 94-02, Dept. Information Science, Univ. Tokyo.

Institute of Electrical and Electronic Engineers, Inc. (1995), Information technology - portable operating system interface (posix) - part 1 : System application program interface (api) - amendment 2 : Threads extension [c language], Technical Report Standard 1003.1c-1995, IEEE, New York , N.Y. also ISO/IEC 9945-1 :1990c.

Jones, M. J. (1991), Bringing the C libraries with us into a multi-threaded future, *in* 'USENIX, Winter', Dallas, TX, pp. 81–92.

Kaashoek, M. F. (1992), Group Communication in Distributed Computer Systems, PhD thesis, Vrije Universiteit Amsterdam.

Kale, L. V. (1994), Parallel programming with CHARM : An overview, Technical report, Univ. of Illinois.

Kale, L. V. & Krishnan, S. (1993), CHARM++ : A portable concurrent object oriented system based on C++, *in* 'Proc. of OOPSLA-93', Vol. 28 of *ACM Sigplan Notes*, Washington, DC, pp. 91–108. Also Tech. Report UIUCDCS-R-93-1796, March 93, University of Illinois, Urbana, IL.

Kannat, S. E., Morel, E., Kitajima, J. P. & Briat, J. (1994), A platform to study dynamic load balancing functions for parallel logic systems, *in* 'IEEE First International Workshop on Parallel Processing', Bangalore - India.

Keppel, D. (1993*a*), Register windows and user-space threads on the sparc, Tech. Rep. UWCSE-93-05-06, Univ. of Washington.

Keppel, D. (1993*b*), Tools and techniques for building fast portable threads packages, Tech. Report UWCSE 93-05-06, University of Washington.

Kernighan, B. W. & Ritchie, D. M. (1978), *The C Progamming Language*, Prentice Hall.

Kleiman, S. R. & Eykholt, J. R. (1995), 'Interrupts as threads', *ACM Operating Systems Review*.

Kleiman, S., Shah, D. & Smaalders, B. (1996), *Programming with thread*, SunSoft Press - Prentice Hall.

Knuth, D. (1973), *The art of computer programming, Volume 1 : Fundamental Algorithms*, Addison Wesley.

Konstandinidou, S. & Upfal, E. (1991), Experimental comparison of multistage networks, Technical Report RJ-8451, IBM, Almaden Research Center.

Krone, O., Aguilar, M. & Hirsbrunner, B. (1995), PT-PVM : Using PVM in a multi-threaded environment, *in* J. Dongarra, M. Gengler, B. Tourancheau & X. Vigouroux, eds, 'Proc. Second European PVM User's Group Meeting', Hermes, Ecole Nationale Supérieure, Lyon, France, pp. 83–88.

Kuntz, J.-M. (1993), Evaluation de la performance d'architectures à base de mémoires caches pour des systèmes multiprocesseurs, PhD thesis, Strasbourg.

L.Albinson, D.Grabas, P.Piovesan, M.Trombroff, C.Tricot & H.Yassaie (1991), UNIX on a loosely coupled architecture : the CHORUS/MIX approach, Tech. Rep. CS/TR-91-49, Chorus systèmes, France.

Lampson, B. W. & Redell, D. D. (1980), 'Experience with processes and monitors in Mesa', *Comm. ACM* **23**(2), 105–117.

Ledgard, H. (1982), *ADA : une introduction*, Masson. ("ADA : an introduction", Springer-Verlag).

Leffler, S. J., Mackusick, M. K., Karels, M. J. & Quaterman, J. S. (1989), *The desgin and implementation of the 4.3 BSD UNIX operating system*, Addison-Wesley.

Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M. & S.Lam, M. (1992), 'The stanford dash multiprocessor', *Computer* **25**(3), 63–79.

Li, K. & Hudak, P. (1989), 'Memory coherence in shared virtual memory systems', *ACM Trans. on Comp. Syst.* **7**(4), 321–359.

Littman, J. (1992), Applying threads, *in* 'Proc. USENIX Conference', pp. 209–222.

Manchek, R. J. (1994), Design and implementation of PVM version 3, Master thesis, University of Tennessee, Knoxville, Tennessee.

Maquelin, O., Gao, G. R., Hum, H. H. J., Theobald, K. B. & Tian, X.-M. (1996), Polling-watchdog : Combining polling and interrupts for efficient message passing, *in* 'Proc. 23rd. Ann. Intl. Symp. on Computer Architecture', ACM, Philadelphia, Pensylvania, pp. 179–188.

Marchal, B. (1996), Un nouveau mode d'auto-ordonnancement : l'auto-ordonnancement par bande, Master's thesis, LMC, IMAG, Grenoble, France.

Marsh, B. D., Scott, M. L., Leblanc, T. J. & Markatos, E. P. (1991), 'First class user-level threads', *ACM Operating System Review* **25**, 483–490.

M.Braner (1988), *Trollius Manuals*, Cornell Theory Center.

McJones, P. R. & Swart, G. F. (1987), Evolving the UNIX system interface to support multithreaded programs, Tech. rep., Digital Systems Research Center. also *Proc. Winter 1989 USENIX Conference, Jan. 89.*

Mehrotra, P. & Haines, M. (1994), An overview of the Opus language and runtime system, *in* 'Proc. of the 7th Annual Workshop on Languages and Compilers for Parallel Computers, LNCS 892', New York. also ICASE Tech.Rep. 94-39.

Message Passing Interface Forum (1994), MPI : A message passing interface standard, Technical Report CS-94-230, Computer Science Dept., University of Tennessee, Knoxville, TN. also in International Journal of Supercomputer Applications, Vol. 8 (3/4), 1994.

Message Passing Interface Forum (1995), MPI-2 : Extensions to the message passing interface, Technical report, University of Tennessee, Knoxville, Tennessee.

Kuntz, J.-M. (1993), Evaluation de la performance d'architectures à base de mémoires caches pour des systèmes multiprocesseurs, PhD thesis, Strasbourg.

L.Albinson, D.Grabas, P.Piovesan, M.Trombroff, C.Tricot & H.Yassaie (1991), UNIX on a loosely coupled architecture : the CHORUS/MIX approach, Tech. Rep. CS/TR-91-49, Chorus systèmes, France.

Lampson, B. W. & Redell, D. D. (1980), 'Experience with processes and monitors in Mesa', *Comm. ACM* **23**(2), 105–117.

Ledgard, H. (1982), *ADA : une introduction*, Masson. ("ADA : an introduction", Springer-Verlag).

Leffler, S. J., Mackusick, M. K., Karels, M. J. & Quaterman, J. S. (1989), *The desgin and implementation of the 4.3 BSD UNIX operating system*, Addison-Wesley.

Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M. & S.Lam, M. (1992), 'The stanford dash multiprocessor', *Computer* **25**(3), 63–79.

Li, K. & Hudak, P. (1989), 'Memory coherence in shared virtual memory systems', *ACM Trans. on Comp. Syst.* **7**(4), 321–359.

Littman, J. (1992), Applying threads, *in* 'Proc. USENIX Conference', pp. 209–222.

Manchek, R. J. (1994), Design and implementation of PVM version 3, Master thesis, University of Tennessee, Knoxville, Tennessee.

Maquelin, O., Gao, G. R., Hum, H. H. J., Theobald, K. B. & Tian, X.-M. (1996), Polling-watchdog : Combining polling and interrupts for efficient message passing, *in* 'Proc. 23rd. Ann. Intl. Symp. on Computer Architecture', ACM, Philadelphia, Pensylvania, pp. 179–188.

Marchal, B. (1996), Un nouveau mode d'auto-ordonnancement : l'auto-ordonnancement par bande, Master's thesis, LMC, IMAG, Grenoble, France.

Marsh, B. D., Scott, M. L., Leblanc, T. J. & Markatos, E. P. (1991), 'First class user-level threads', *ACM Operating System Review* **25**, 483–490.

M.Braner (1988), *Trollius Manuals*, Cornell Theory Center.

McJones, P. R. & Swart, G. F. (1987), Evolving the UNIX system interface to support multithreaded programs, Tech. rep., Digital Systems Research Center. also *Proc. Winter 1989 USENIX Conference, Jan. 89.*

Mehrotra, P. & Haines, M. (1994), An overview of the Opus language and runtime system, *in* 'Proc. of the 7th Annual Workshop on Languages and Compilers for Parallel Computers, LNCS 892', New York. also ICASE Tech.Rep. 94-39.

Message Passing Interface Forum (1994), MPI : A message passing interface standard, Technical Report CS-94-230, Computer Science Dept., University of Tennessee, Knoxville, TN. also in International Journal of Supercomputer Applications, Vol. 8 (3/4), 1994.

Message Passing Interface Forum (1995), MPI-2 : Extensions to the message passing interface, Technical report, University of Tennessee, Knoxville, Tennessee.

Otte, R., Patrick, P. &.Roy, M. (1996), *Understanding CORBA : the common object request broker architecture*, Prentice Hall.

Papka, M. (1995), The CAVEcomm library, Tech.rep., ANL.

Parasoft Corporation (1988), *Express version 1.0 : A Communication Environment for Parallel Computers.*

Parasoft Corporation (1992), *Express User's Guide*, version 3.2.5 edn, Parasoft Corporation, Pasadena, CA.

Peacock, J., Saxena, S., Thomas, D., Yang, F. & Yu, W. (1992), Experiences from multithreading System V Release 4, *in* 'Proc. USENIX SEDMS III Symp.', pp. 77–92.

Pierce, P. (1988), The NX/2 operating system, *in* ACM Press, ed., 'Proc. of the Third Conference on Hypercube Concurrent Computers and Applications', pp. 384–390.

Pinakis, J. (1992), Remote thread execution, Technical report, DCS, Univ. of Western Australia.

Plateau, B. & al. (1993), Présentation d'APACHE, Rapport APACHE 1, IMAG, Grenoble.

PORTS Consortium (1995), *the PORTS0 Interface.*

Postel, J. (1981*a*), Transmission Control Protocol, Technical Report RFC 793, Information Sciences Institute.

Postel, J. (1981*b*), User Datagram Protocol, Technical Report RFC 768, Information Sciences Institute.

Pountain, D. (1986), *A tutorial introduction to Occam programming*, Inmos, Colorado Springs, Co.

Powell, M. L., Kleiman, S. R., Barton, S., Shah, D., Stein, D. & Weeks, M. (1991), SunOs 5.0 multithread architecture, *in* 'Proc. of the Winter 1991 USENIX Conf.', pp. 65–79.

Puaut, I., Banâtre, M. & Routeau, J.-P. (1991), Early experience with building and using the gothic distributed operating system, *in* 'Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II)', USENIX, Atlanta GA, pp. 271–282.

Raina, S. (1992), Virtual shared memory : a survey of techniques and systems, Tech. Rep. CSTR-92-36, DCS, Univ. Bristol, UK.

Reed, D. P. & Kanodia, R. K. (1979), 'Synchronization with event counts and sequencers', *Comm. ACM* **22**(2), 115–123.

R.Namyst & J-F.Méhaut (1995), PM2 : Parallel multithreaded machine, a computing environment for distributed architectures, *in* 'Proc. ParCo'95, Gent, Belgium'.

Roch, J. L., Vermeerbergen, A. & Villard, G. (1994), A new load-prediction scheme based on algorithmic cost functions, *in* Springer-Verlag, ed., 'Proc. of CONPAR 94-VAPP VI', LNCS 854, Linz Austria.

Rosenberry, W. & Teague, J. (1993), *Distributing Applications across DCE and Windows NT*, O'Reilly & Associates, Inc. ISBN 1-56592-047-3.

Rosing, M. & Saltz, J. (1993), Low latency messages on distributed memory multiprocessors, Tech. Report ICASE 93-30, Institute for Computer Applications Science and Engineering, NASA LaRC, Hampton, Virginia.

Rovner, P., Levin, R. & Wick, J. (1985), On extending modula-2 for building large, integrated systems, Tech. Rep. 3, DEC Systems Research Center.

Sakai, S., Okamoto, K., Matsuoka, H., Hirono, H., Kodama, Y. & Sato, M. (1993), Super-threading : Architectural and software mechanisms for optimizing parallel computation, *in* 'Proc. 1993 International Conf. on Supercomputing', Tokyo, pp. 251–260.

Schmidtmann, C., Tao, M. & Watt, S. (1993), Design and implementation of a multithreaded Xlib, *in* 'Winter USENIX', San Diego, CA, pp. 193–203.

Schroeder-Preikschat, W. (1990), PEACE - a distributed operating system for high-performance multicomputer systems, *in* 'LNCS 433', pp. 23–44.

Seevers, B., Quinn, M. J. & Hatcher, P. J. (1992), A parallel programming environment supporting multiple data-parallel modules, *in* 'Workshop on Languages, Compilers and Run-time Environments for Distributed Memory Machines'.

Shirley, J. (1992), *Guide to writing DCE Applications*, O'Reilly & Associates, Inc. ISBN 1-56592-004-X.

Shu, W. (n.d.), *Runtime Support for User-Level Ultra Lightweight Threads on Massively Distributed Memory Machines*, Depart. of Computer Science, State University of New York at Buffalo.

Skjellum, A., Doss, N. E. & Vishwanathan, K. (1994*a*), Inter-communicator extensions to MPI in the MPIX (MPI eXtension) library, Technical report, CSD and NSF ERC, Mississippi State University.

Skjellum, A., Doss, N. E., Viswanathan, K., Chowdappa, A. & Bangalore, P. V. (1994*b*), Extending the message passing interface (MPI), *in* 'Proc. 1994 Scalable Parallel Libraries Conf.', IEEE Computer Society Press. also CSD and NSF ERC, Mississippi State University Tech.Rep.

Snir, M. (1992), Scalable parallel computers and scalable parallel codes : from theory to practice, *in* Springer-Verlag, ed., 'Parallel Architecures and Their Efficient Use. LNCS 678', pp. 176–184.

Souza, R. J., Krishnakumar, P. G., Ozveren, C. M., Simcoe, R. J., Spinney, B. A., Thomas, R. E. & Walsh, R. J. (1994), 'GIGAswitch system : A high-performance packet-switching platform', *Digital Technical Journal* **6**(1), 9–22.

Spertus, E., Goldstein, S. C., Schauser, K. E., von Eicken, T., Culler, D. E. & Dally, W. J. (1993), Evaluation of mechanisms for fine-grained parallel programs in the J-Machine and the CM-5, *in* 'Proc. 20th Ann. Intl. Symp. on Computer Architectures', San Diego, Ca, pp. 302–313.

Stein, D. & Shah, D. (1992), Implementing lighweight threads, *in* 'Proc. of the summer 1992 USENIX Conf.', pp. 1–9.

Sternström, P. (1990), 'A survey of cache coherence schemes for multiprocessors', *Computer* **23**(6), 12–24.

Stevens, W. R. (1990), *UNIX network programming*, Prentice-Hall.

Stroustrup, B. (1991), *The C++ Programming Language (2nd Edition)*, Addison Wesley.

Stunkel, C. B., Shea, D. G., Grice, D. G., Hochschild, P. H. & Tsao, M. (1994), 'The SP1 High-Performance Switch', *IEEE* pp. 150–157.

Sun Microsystems Inc. (1986), *Remote Procedure Call Programming Guide*, Sun Microsystems, Inc., Mountain View, CA.

Sun Microsystems Inc. (1987*a*), *Lightweight Process Library Manual Pages*, release 4.0 edn, Sun Microsystems Inc.

Sun Microsystems Inc. (1987*b*), XDR : External Data Representation standard, Technical Report RFC 1014, Sun Microsystems Inc.

Sun Microsystems Inc. (1988), 'RPC: Remote procedure call protocol specification version 2 ;RFC1058', *Internet Request for Comments*.

Sun Microsystems Inc. (1990), *Lightweight Process Library*, release 4.1 edn, Sun Microsystems Inc.

Sundaresan, N. & Lee, L. (1994), An object-oriented thread model for parallel numerical applications, *in* 'Proc. 2nd Annual Object-Oriented Numerics Conf.', Sunriver, OR, pp. 291–308.

Sunderam, V. (1990), 'PVM : A framework for parallel distributed computing', *Concurrency : Practice and Experience* **2**(4), 315–339.

Sunsoft, Inc. (1992), 'Multithreading in SunOS', *Software Technical Bulletin*.

Tanenbaum, A. S. (1992), *Modern Operating Systems*, Prentice Hall.

T.Bubeck, M.Hiller, W.Küchlin & R.Rosentiel (1995), Distributed symbolic computation with DTS, *in* A.Ferreira & J.Rolim, eds, 'Proc. 2nd Int. Workshop, IRREGULAR'95, Parallel Algorithms for Irregularly Structured Problems, LNCS 980', Lyon, France.

Teodorescu, F. & de Kergommeaux, J. C. (1995), Performance evaluation of parallel programs by multiple phases of execution, *in* 'Romania Open Systems Event ROSE'95', Bucharest, Romania.

Tevanian, A., Rashid, R. F., Golub, D. B., Black, D. L., Cooper, E. & Young, M. W. (1987), Mach threads and the Unix kernel : the battle for control, *in* 'USENIX', pp. 185–197.

T.Muntean (1994), A generic virtual machines architecture for distributed parallel operating systems design, *in* 'Proc. ACM-IEEE Int. Parallel Processing Symp. IPPS'94', Cancun, Mexico.

T.Muntean, A.Balaniuk, H.Castro, R.Despons, A.Elleuch, F.Menneteau, L.Mugwaneza, E-G.Talbi & Ph.Waille (1993), PAROS : A generic multi virtual machines parallel operating system, *in* G. Joubert, D.Trystram, F.J.Peters & D.J.Evans, eds, 'Parallel Computing : Trends and Applications, Proc. Parco'93, Grenoble, France', pp. 319–328.

Tristram, C. (1995), 'Spec 1170-based Unix', *Open Computing* **12**(3), 65–.

Tsobgny, Y. L. (1991), PARX :Architecture de noyau de système d'exploitation parallèle, PhD thesis, Institut National Polytechnique de Grenoble, France.

Tubtiang, A. (1993), Un commutateur ATM pour le réseau numérique à intégration de service large bande, PhD thesis, Paris 6.

Ullman, J. (1984), *Computational Aspects of VLSI*, Computer Science Press.

Stunkel, C. B., Shea, D. G., Grice, D. G., Hochschild, P. H. & Tsao, M. (1994), 'The SP1 High-Performance Switch', *IEEE* pp. 150–157.

Sun Microsystems Inc. (1986), *Remote Procedure Call Programming Guide*, Sun Microsystems, Inc., Mountain View, CA.

Sun Microsystems Inc. (1987*a*), *Lightweight Process Library Manual Pages*, release 4.0 edn, Sun Microsystems Inc.

Sun Microsystems Inc. (1987*b*), XDR : External Data Representation standard, Technical Report RFC 1014, Sun Microsystems Inc.

Sun Microsystems Inc. (1988), 'RPC: Remote procedure call protocol specification version 2 ;RFC1058', *Internet Request for Comments*.

Sun Microsystems Inc. (1990), *Lightweight Process Library*, release 4.1 edn, Sun Microsystems Inc.

Sundaresan, N. & Lee, L. (1994), An object-oriented thread model for parallel numerical applications, *in* 'Proc. 2nd Annual Object-Oriented Numerics Conf.', Sunriver, OR, pp. 291–308.

Sunderam, V. (1990), 'PVM : A framework for parallel distributed computing', *Concurrency : Practice and Experience* **2**(4), 315–339.

Sunsoft, Inc. (1992), 'Multithreading in SunOS', *Software Technical Bulletin*.

Tanenbaum, A. S. (1992), *Modern Operating Systems*, Prentice Hall.

T.Bubeck, M.Hiller, W.Küchlin & R.Rosentiel (1995), Distributed symbolic computation with DTS, *in* A.Ferreira & J.Rolim, eds, 'Proc. 2nd Int. Workshop, IRREGULAR'95, Parallel Algorithms for Irregularly Structured Problems, LNCS 980', Lyon, France.

Teodorescu, F. & de Kergommeaux, J. C. (1995), Performance evaluation of parallel programs by multiple phases of execution, *in* 'Romania Open Systems Event ROSE'95', Bucharest, Romania.

Tevanian, A., Rashid, R. F., Golub, D. B., Black, D. L., Cooper, E. & Young, M. W. (1987), Mach threads and the Unix kernel : the battle for control, *in* 'USENIX', pp. 185–197.

T.Muntean (1994), A generic virtual machines architecture for distributed parallel operating systems design, *in* 'Proc. ACM-IEEE Int. Parallel Processing Symp. IPPS'94', Cancun, Mexico.

T.Muntean, A.Balaniuk, H.Castro, R.Despons, A.Elleuch, F.Menneteau, L.Mugwaneza, E-G.Talbi & Ph.Waille (1993), PAROS : A generic multi virtual machines parallel operating system, *in* G. Joubert, D.Trystram, F.J.Peters & D.J.Evans, eds, 'Parallel Computing : Trends and Applications, Proc. Parco'93, Grenoble, France', pp. 319–328.

Tristram, C. (1995), 'Spec 1170-based Unix', *Open Computing* **12**(3), 65–.

Tsobgny, Y. L. (1991), PARX :Architecture de noyau de système d'exploitation parallèle, PhD thesis, Institut National Polytechnique de Grenoble, France.

Tubtiang, A. (1993), Un commutateur ATM pour le réseau numérique à intégration de service large bande, PhD thesis, Paris 6.

Ullman, J. (1984), *Computational Aspects of VLSI*, Computer Science Press.

Univ. Illinois (1992), *The CHARM (3.2) Programming Language Manual*, University of Illinois, Urbana Champaign, IL.

Upfal, E. (1989), An o(log n) deterministic packet routing scheme, *in* 'Proc. 21st annual Symposium on Theory of Computing'.

Vermeerbergen, A. (1994), Les poly-algorithmes et la prévision de coûts pour une expression portable et extensible du parallélisme, *in* L. Bougé, ed., 'Actes des 6èmes Rencontres Francophones du Parallélisme, RenPar'6', Ecole Normale Supérieure, Lyon, France, pp. 51–54.

von Eicken, T. & Culler, D. E. (1992), 'Building communication paradigms with the CM-5 Active Message layer (CMAM)'.

von Eicken, T., Culler, D. E., Goldstein, S. C. & Schauser, K. E. (1992), Active messages : A mechanism for integrated communication and computation, *in* 'Proceedings of the 19th Annual International Symposium on Computer Architecture', Gold Coast, Australia, pp. 256–266. Also Tech. Report UCB-CSD 92-675, Computer Science Division, EECS, University of California, Berkeley, CA.

Wallach, D. A., Hsieh, W. C., Johnson, K., Kaashoek, M. F. & Weihl, W. E. (1995), Optimistic active messages : a mechanism for scheduling communication with computation, Tech. rep., LCS, MIT.

wei H. Lehman, L. (1993), 'Integrating Zipcode and PVM : Towards a higher-level message-passing environment'.

Weihl, W. E. (1989), *Distributed Systems*, ACM Press, chapter 4 : Remote Procedure Call, pp. 65–86.

Zhou, H. & Geist, A. (1995*a*), LPVM : A step towards multithread PVM, Technical report, MCS, ORNL.

Zhou, H. B. & Geist, A. (1995*b*), Faster message passing in PVM, *in* 'Proc. 9th Int. Parallel Processing Symposium : Workshop on High-Speed Network Computing'.

Z.Lahjmri & T.Priol (1992), 'Koan : a shared memory for the iPSC/2 hypercube', *Lecture Notes in Computer Science* **634**, 44–452.

Résumé

Nous présentons un support d'exécution pour applications parallèles irrégulières. Par le terme irrégulier nous entendons des applications dont le comportement ne peut pas être prévu indépendamment du problème effectif à résoudre. En conséquence, le calcul d'un « bon » ordonnancement pour de telles applications est impossible. Il est alors nécessaire de permettre l'exécution dynamique et concurrente d'un grand nombre de calculs de grain éventuellement fin, et ce avec un coût minimum pour ne pas grever l'efficacité.

L'approche retenue dans le cadre du projet APACHE consiste, pour assurer la portabilité efficace des applications, à exploiter le concept de poly-algorithme et à l'exprimer à l'aide d'une décomposition procédurale parallèle. L'opérateur de base de notre support d'exécution, l'appel de procédure à distance asynchrone, permet d'exprimer une telle décomposition procédurale. Cet opérateur est réalisé par le couplage lâche d'un noyau de multiprogrammation légère et d'un noyau de communication (PVM). Chaque calcul (exécution d'une procédure) est alors réalisé par un fil d'exécution différent.

Nous décrivons le modèle de programmation que nous avons retenu, les choix de réalisation et l'implantation effectuée. Nous exposons en particulier le problème du couplage de la progression des calculs et de celle des communications, couplage réalisé à l'aide d'une opération « d'ordonnancement-scrutation ». Cette réalisation est ensuite évaluée selon divers critères (portabilité, latence, débit, recouvrement, performances d'une application réelle). Nous présentons en dernier lieu 13 autres supports d'exécution de but semblable : utiliser la multiprogrammation légère pour améliorer le support des applications parallèles de grain variable. Nous tentons en particulier de dégager les grandes lignes de comparaison entre ces exécutifs, et présentons les diverses solutions retenues pour le couplage multiprogrammation légère / communications. Nous terminons par une indication d'un paradigme de programmation plus évolué, extension de la notion de décomposition procédurale parallèle.

Abstract

We investigate run-time support for irregular parallel applications. By irregular we mean that the behaviour can't be foreseen independently of the effective problem to be solved. Consequently, a « good » schedule can't be established for such applications. It is thus necessary to support dynamically and concurrently a large number of threads of computation, with possibly a smaller grain of parallelism. This support must be done at low cost to allow for efficient execution.

The method explored by the APACHE project consists of exploiting the concept of polyalgorithm in order to allow for the efficient portability of applications. This concept is expressed with the help of a parallel procedural decomposition. The basic operator of our run- time support, the asynchronous remote procedure call, permits the expression of such a parallel procedural decomposition. This operator is built by loosely coupling a multithreading kernel and a communication kernel (PVM). Each computation (procedure run) is thus implemented by a different thread of computation.

We describe the programming model that we selected, the implementation choices, and details. In particular we show the problem of coupling the advance of computations and that of communications. This is done with the help of a « scheduling-polling » operation. This implementation is then estimated according to different criteria : portability, latency, throughput, overlap, and real application performances. In a last step we present 13 other run- time systems with similar goals : using multithreading to help the support of variable grain size parallel applications. In particular, we describe the basis of comparison among them, and show the different solutions given to the problem of coupling multithreading and communication. We conclude with a suggestion for a more elaborate programming paradigm, extending the concept of parallel procedural decomposition.