MASTER EN SCIENCES INFORMATIQUES

Équilibrage dynamique de charge pour des calculs parallèles sur cluster Linux une évaluation de l'environnement AMPI

Teller, Frédéric

Award date:
2004

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 2003-2004

Équilibrage Dynamique de Charge pour des Calculs Parallèles sur Cluster Linux - Une Évaluation de l'Environnement AMPI

Frédéric Teller

Mémoire présenté en vue de l'obtention du grade
de Maître en Informatique

Résumé

L'informatique parallèle ou l'utilisation combinée de plusieurs processeurs pour la programmation d'applications scientifiques est devenue courante aujourd'hui. Pour des applications irrégulières, un système d'équilibrage de charge des processeurs est à considérer, afin d'optimiser les performances. Ce mémoire considère cette problématique de régulation dynamique de charge. Il présente une synthèse des différents modèles et environnements de programmation parallèle.
Un environnement spécifique, AMPI (*Adaptiv* MPI), propose des optimisations intelligentes et adaptatives pour l'équilibrage de charge de programmes MPI traditionnels. Cet environnement est spécialement étudié. Des résultats sont donnés pour une application test ainsi qu'une application pratique de dynamique moléculaire. De ses résultats, il ressort qu'AMPI est intéressant pour des applications irrégulières sur une architecture hétérogène comme un cluster non dédié à une application.

MOTS CLÉS : *équilibrage de charge, calcul parallèle, cluster, mpi, ampi, dynamique moléculaire.*

Summary

Parallel computing or the cooperative utilisation of a set of processors for scientific application programming has become current nowadays. For irregular application a load balancing system is necessary in order to optimise performance. This master's thesis is focusing on this particular issue of dynamic load balancing. It presents a synthetic view of some parallel programming models and runtime supports.
A specific system, AMPI (*Adaptiv* MPI), carries out intelligent adaptive optimisation for traditional parallel MPI programs. This system is specially studied. Results are given for a test application and an application experience involving molecular dynamics. These results show that AMPI can be useful for irregular or dynamic applications on an heterogeneous architecture, like a non dedicated cluster.

KEYWORDS : *load balancing, parallel computing, cluster, mpi, ampi, molecular dynamics.*

Avant propos

Initialement ce mémoire était cadré dans le projet de recherche MECCANO (Maximalisation de l'Efficacité des Cluster de Calcul Avec un Nouvel Ordonnanceur) [33] que la région Wallonne n'a finalement pas retenu. Néanmoins, une grande partie du mémoire a été développée sur le sujet initial du mémoire au sein du Service Informatique de la Faculté Polytechnique de Mons, sous la direction de Pierre Manneback qui était l'initiateur du projet.

Mes premiers remerciements s'adressent donc naturellement à Pierre Manneback, pour son accueil chaleureux à Mons et pour son suivi particulier du stage et du mémoire.
Un très grand merci aussi, à mon promoteur Vincent Englebert pour son aide dans la rédaction de ce document, spécialement ses relectures critiques et multiples des différentes versions du mémoire et les nombreuses suggestions constructives de sa part.
Je tiens également à remercier très chaleureusement les membres du laboratoire CRMM (Centre de Recherche en Modélisation Moléculaire) du pôle d'excellence Materia Nova à Mons (Dr. J. De Coninck) pour l'emprunt d'une de leurs simulations physiques pour des tests pratiques et pour leur soutien dans le domaine théorique de la physique en dynamique moléculaire. J'y ajoute une mention spéciale pour Emilie Bertrand, une étudiante de la deuxième licence en sciences physiques de l'UMH, sans laquelle la simulation physique n'aurait pu être réalisée.
L'équipe de développement d'AMPI est également à remercier pour leur interaction personnelle pour l'exploitation de leurs systèmes.
Un grand merci aussi, à Sopheap Seng, un étudiant du DEA inter-universitaire en informatique inscrit à la FPMS, pour ses idées et initiatives qui m'ont aidé dans le développement de ce mémoire.
Une très grande partie de cette rédaction a été revue par Marie-Élisabeth André qui a eu le remarquable courage de corriger mes nombreuses fautes françaises ; je l'en remercie.
Enfin, c'est vers ma copine Astrid que j'adresse mes remerciements les plus tendres pour toutes ces longues journées qu'elle a dû patienter sans m'avoir à ses cotés.

Table des matières

Introduction

Le concept de "Grid Computing" ou infrastructure informatique hétérogène de calcul à haute performance est très étudié actuellement. Comme noeud de ces grilles, on trouve notamment des serveurs parallèles. Les problèmes d'équilibrage de charge (*load balancing*) et d'ordonnancement des tâches (*job scheduling*) sont évidemment importants afin de maximaliser l'utilisation des ressources et de minimaliser les temps de réponses.

Dans ce mémoire, nous avons évalué d'un point de vue académique des ordonnanceurs dynamiques et adaptatifs pour des calculs parallèles sur des clusters Linux. Pour cela, nous avons parcouru une série d'environnements parallèles utilisables sur un réseau d'ordinateur.
Du côté pratique, le travail se base principalement sur les travaux menés dans le laboratoire de programmation parallèle de l'université d'Illinois[1]. Ce laboratoire a développé une implémentation du standard MPI (Message Passing Interface), à savoir l'environnement AMPI (Adaptative MPI). Particulièrement, nous nous sommes intéressés pour les capacités d'équilibrage dynamique de charge (*dynamic load balancing)* d'AMPI.

Ce document est structuré en six chapitres qui concernent des objectifs différents. Après une courte citation des objectifs poursuivis dans ce mémoire le premier chapitre donne une introduction dans le domaine de l'informatique parallèle. Ainsi le lecteur "non expert" peut se familiariser avec les concepts généraux du calcul parallèle. Notamment les trois mots clés du titre de ce mémoire sont investigués à savoir : les clusters sous forme d'architecture parallèle, les concepts de l'équilibrage de charge, et les paradigmes de programmation parallèle. Spécialement, les concepts clés de ce mémoire sont repris dans un glossaire à la fin de ce document.
Le deuxième chapitre présente, à partir des exigences d'un système parallèle, l'état d'art sur les environnements de programmation parallèles supportant l'équilibrage dynamique de charge. Pour continuer, au chapitre 3 avec une évaluation académique des différentes technologies proposées par rapport aux différents objectifs fixés au début et en retenant finalement l'environnement AMPI pour des tests pratiques.
Les chapitres 4 et 5 montrent respectivement une comparaison entre MPI et AMPI sur un exemple test de type académique et sur une application d'un cas réel d'une simulation du domaine physique en dynamique moléculaire. Le premier de ces deux chapitres montre bien les différents concepts et propriétés que l'environnement AMPI fournit. Le chapitre 5 est sûrement le chapitre le plus intéressant au niveau apport original de ce mémoire. Il montre notamment la conception d'un programme parallèle MPI d'une simulation en dynamique moléculaire à partir d'une source séquentielle, puis la transformation de ce programme MPI en un programme AMPI, pour terminer avec différentes mesures et leurs interprétations sur des résultats et sur des notions d'efficacité que les approches parallèles réclament.

[1] Parallel Programming Laboratory, Dept of Computer Science, University of Illinois, Urban Champaign (voir `http://charm.cs.uiuc.edu`).

Finalement, ce document se termine avec la conclusion sur l'utilité de l'environnement AMPI pour des applications irrégulières dans un contexte hétérogène ou dynamique. Une autre remarque concerne le fait que les principes académiques ne sont pas toujours facilement applicables et transposables dans chaque contexte réel. En effet, les résultats des tests pratiques effectués avec notre application réelle montrent parfois des particularités à la fois intéressantes et surprenantes.

Objectifs du mémoire

Des algorithmes et logiciels parallèles pour le calcul nécessitent de très hautes performances, étant donné le volume d'opérations à effectuer, ou en raison des contraintes de temps-réel. Certains logiciels de simulation nécessitent en effet des quantités énormes de calcul, requérant ainsi la disponibilité de très gros serveurs de calcul et des temps d'exécution fort longs. L'utilisation d'ordinateurs multi-processeurs, de clusters de PCs ou des nouveaux concepts de "Grid Computing" est la voie idéale (à la fois en terme de coût et de performance) pour faire tourner ces applications gourmandes en temps.
Vu l'intérêt de l'informatique parallèle, **un premier objectif** de ce mémoire est de donner une introduction globale dans ce domaine. Rendant ainsi le mémoire accessible aux lecteurs qui ne sont pas issus du domaine.

Dans des calculs parallèles la charge de calcul d'un processeur dépend du nombre des tâches allouées au processeur, de la complexité mathématique pour calculer ces tâches, ou encore de la taille de ces tâches. Si en plus, on considère que le travail pour calculer une tâche peut évoluer dynamiquement pendant l'exécution (par exemple dans des simulations physiques), que le réseau de calcul peut être hétérogène (machines aux puissances différentes, avec des tailles et vitesses de mémoires différentes et/ou connectées de façon non uniforme) ou que la distribution de charge sur ce réseau est non uniforme (par exemple l'utilisation d'une machine pour des applications interactives), le besoin d'un système de répartition adaptative de charge selon la demande devient nécessaire pour garantir de bonnes performances et la bonne utilisation des ressources.
Le deuxième objectif central de ce document est alors une étude et évaluation des systèmes de l'équilibrage dynamique de charge (*dynamic load balancing*) et des problèmes de l'ordonnancement (*scheduling*) pour des calculs parallèles. En effet, les problèmes du calcul parallèle sont déjà bien examinés depuis leur début dans les années quatre-vingts et sont bien établis dans des standards comme MPI [35] ou des langages comme PVM [42] ou le HPF [23]. Par contre, le domaine d'équilibrage dynamique des charges est plus récent. Ces concepts théoriques bien établis depuis la fin des années nonante trouvent dans les dernières années de plus en plus des implémentations pratiques. Ce mémoire va donc examiner certaines de ces implémentations. Pour restreindre le domaine d'investigation nous avons choisi, pour une partie pratique, une architecture cible facilement disponible de nos jours, à savoir un cluster de PCs. De plus, vu la base de ce mémoire dans l'informatique parallèle, nous nous sommes intéressés à des environnements d'équilibrage dynamique qui utilisent des concepts de calcul parallèle bien établis comme le standard MPI(Message Passing Interface). Spécialement, AMPI (Adaptative MPI) [1] respecte ces contraintes et est, en fait, une implémentation du standard MPI avec des capacités d'équilibrage dynamique sur un cluster de PCs.

Pour ne pas se limiter à un cadre uniquement théorique, l'environnement AMPI a été testé pratiquement avec un exemple test de type académique ainsi qu'avec une application réelle du domaine physique de la dynamique moléculaire. **Le troisième objectif**

important qui offre aussi l'apport le plus original de ce mémoire concerne donc une étude de cas d'AMPI avec une simulation empruntée et adaptée du domaine de la dynamique moléculaire. Ce troisième but essaye donc de montrer les défis et difficultés d'AMPI dans un cadre réel.

Chapitre 1

Notions et concepts fondamentaux de l'informatique parallèle

Ce premier chapitre a pour but d'introduire le lecteur dans le domaine de l'informatique parallèle, notamment, dans les trois concepts clés du titre de ce document, à savoir l'équilibrage de charge dynamique, le calcul parallèle et les cluster Linux, qui sont replacés dans leur contexte théorique. Nous allons élaborer et expliquer les concepts fondamentaux sous-jacents de ces trois points.
Globalement, nous allons donc analyser les notions d'architectures parallèles, d'équilibrage de charge et les différents paradigmes et théories utilisés pour le calcul parallèle.

1.1 Différentes architectures parallèles

L'évolution des architectures parallèles au fil du temps a produit des machines dont les puissances de calcul croissent de plus en plus vite. Un simple moyen de comprendre les différents types d'architecture qui sont ou qui ont été à la base de ces machines puissantes est de comparer leurs principales caractéristiques. Pour cela, nous allons parcourir dans cette section deux classifications complémentaires de la littérature. La première est la classification de Flynn qui classifie les architectures selon une approche théorique. Par contre, la seconde classification identifie les différentes architectures parallèles selon l'interconnexion de leurs composants.

1.1.1 Classification de Flynn

La classification de Flynn [16] est probablement la classification la plus ancienne des architectures parallèles. Elle se trouve dans presque chaque livre ou article qui traite ce domaine. Le gros avantage, à coté de la généralité de cette classification, est sa simplicité au niveau théorique. Cependant, il est difficile de faire la relation avec des architectures réelles.

Cette classification distingue les architectures selon la nature de leurs performances. Les architectures parallèles sont classées selon deux axes où l'un est caractérisé par le flux d'instructions et l'autre par le flux de données. Nous avons donc une matrice qui classifie les architectures selon le nombre (unique ou multiple) d'instructions possibles d'effectuer à un moment donné et le nombre (unique ou multiple) de données qui peuvent être traitées en même temps. Sur cette matrice nous identifions alors quatre architectures différentes (voir figure 1.1 [44]).

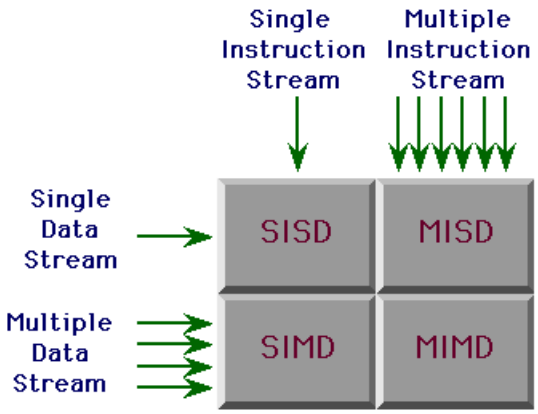


FIG. 1.1 – *Schéma d'une classification par Flynn* [44]

- **SISD (Single Instruction Single Data) :** Cette première architecture n'est pas une architecture parallèle et correspond à un ordinateur classique séquentiel de type *von Neumann*. Nous le citons par souci d'être complêt.
- **SIMD (Single Instruction Multiple Data) :** C'est une architecture qui exécute en parallèle une même instruction sur des données différentes. On applique donc un parallélisme au niveau des données. Un sous-ensemble de cette classe est formé de machines vectorielles. Les architectures SIMD existent de moins en moins en réalité.

- **MISD (Multiple Instruction Single Data) :** Cette classe ne correspond pas a une architecture réel. Néanmoins, elle est comparable à une approche de type *pipeline*.
- **MIMD (Multiple Instruction, Multiple Data) :** C'est la classe la plus importante en terme parallèle. Une architecture de ce type combine les deux approches précédentes, donc traite à la fois plusieurs flots d'instructions sur plusieurs flots de données. Comme exemple de cette classe nous pourrons citer une architecture multiprocesseur.

Un autre grand désavantage de la classification originale de Flynn est qu'elle ne prend pas en compte l'organisation de la mémoire dans une architecture. Nous avons des architectures réelles différentes, selon que l'organisation de la mémoire se fait de manière partagée (*shared memory* **SM**) ou distribuée (*distributed memory* **DM**). Pour cela, différents auteurs [20] [24] ont proposé de diviser les classes de Flynn en deux sous-classes selon l'organisation de la mémoire utilisée. Un autre problème du moyen d'interconnexion entre les processeurs et/ou le(les) mémoire(s) subsiste cependant dans cette classification élargie.

Dans ce mémoire nous allons nous baser sur un modèle d'architecture qui est dérivé de la classe MIMD de Flynn. Cette classe supplémentaire porte l'acronyme **SPMD (Single Program, Multiple Data)**. Elle correspond plus à la réalité actuelle, d'après laquelle les données sont distribuées et traitées par les copies d'un même programme sur différents processeurs. Nous avons donc un parallélisme au niveau du programme. Nous arrivons à des exécutions différentes d'un même programme en le dotant d'un dispositif qui identifie chaque instance du programme initial sur les différents processeurs. En effet, si un programme est capable de s'identifier parmi un ensemble de programmes identiques, il peut, au travers d'un test sur son identité, exécuter le code qui lui est propre (voir page 19).

Exemple : *Programme SPMD pour une implémentation maître-esclave*
SI je_suis_maître
ALORS exécuter_code_maître
SINON exécuter_code_esclave

1.1.2 Une autre taxonomie

La difficulté de transposer la classification de Flynn dans des architectures physiques réelles justifie l'investigation d'une deuxième taxonomie. Cette deuxième approche va donner une vue complémentaire à celle de Flynn et classifie les architectures selon les interconnexions des processeurs et des mémoires. Quatre classes souvent identifiées sont :

- **SMP** *Symmetric Multiprocessors (share-everything)*
  Une architecture SMP est une machine multi-processeurs d'un ensemble de processeurs identiques qui se partagent physiquement la mémoire ainsi que les entrées et sorties (voire figure 1.2 [26]). Ces machines qui accèdent avec la même vitesse à toutes les zones de la mémoire sont aussi appelées machines UMA (*Uniform Memory Acces*).
  Une machine SMP possède en général un système d'exploitation unique qui gère toute l'architecture. Mais, au contraire des ordinateurs parallèles traditionnels[1], les machines SMP ne requièrent pas un système d'exploitation propre à l'architecture. En effet il existe aujourd'hui des OS ordinaires comme Linux, Windows NT ou Sun Solaris pour des SMPs. Souvent ces systèmes d'exploitation simulent le principe d'un système à image unique, c'est à dire qu'ils représentent pour l'utilisateur l'architecture complexe d'un SMP comme un simple ordinateur classique.
  A coté des processeurs identiques, ces architectures possèdent généralement des composants standards. Un SMP actuel a donc un faible coût par rapport aux ordinateurs parallèles traditionnels et est facilement extensible en ajoutant des processeurs.
  Vu que le système d'exploitation est unique et que la coordination (communication, synchronisation, ...) se fait via la mémoire partagée entre les processeurs, une machine SMP est facile à programmer. Une attention particulière doit être accordée aux problèmes de concurrence, comme dans la mémoire partagée, l'accès simultané aux mêmes données est possible. Un grand désavantage d'un SMP est sa limite d'extension. En effet, nous ne pouvons pas augmenter le nombre de processeurs indéfiniment car ils accèdent tous à la même mémoire. Une mémoire commune peut donc assez vite devenir un un élément *bottleneck*.
  Des exemples concrets sont par exemple les Cray J90 ou Entreprise 10000 de Sun.
- **MPP** *Massively Parallel Processors (share-nothing)*
  Dans une architecture multi-processeurs de type MPP, au contraire des machines SMP, les processeurs ne se partagent ni une mémoire unique ni des entrées et sorties (voir figure 1.3 [26]). Chaque processeur a sa propre mémoire et possède une interconnexion rapide avec les autres processeurs. C'est donc une architecture de type MIMD avec une mémoire distribuée.
  Chaque processeur d'un MPP a son système d'exploitation propre, qui rend plus difficile la réalisation d'un système à image unique. L'utilisation de composants standards résulte dans une bonne relation coût/performance. Les machines MMP n'ont pas de limite sur le nombre de processeurs et sont facilement extensibles. Par contre, vu le manque d'un système exploitation à image unique, leur programmation est plus difficile que celle des machines SMP. Ainsi, la communication et coordination est explicite entre les différents processeurs.

[1] Par exemple les machines vectorielles.

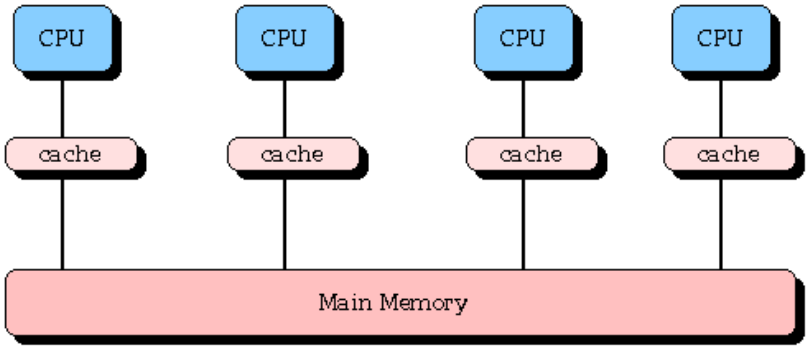


FIG. 1.2 – *Schéma d'une architecture Symmetric Multiprocessors (SMP)*[26].

Des exemples concrets sont par exemple les Cray T3E ou les Cray X1[2].
Pour combler les déficits d'une mémoire distribuée des architectures MPP, une évolution concerne maintenant la création de machines parallèles avec une mémoire virtuellement partagée (*Distributed Shared Memory* DSM). Contrairement aux SMPs, l'accès à la mémoire virtuellement commune, dépend alors de la localisation physique du processeur et de la mémoire dans la machine parallèle. On parle de machine NUMA (*Non Uniform Memory Access*). Mais, pour ne pas compliquer le travail du programmeur, il faut aussi que les différents niveaux de mémoire cache des processeurs, soient synchronisés avec les adresses mémoire qu'ils représentent. On parle alors de machine ccNUMA (*Cache Coherency NUMA*). Pour des raisons de performance, on place souvent dans de tels réseaux, à la place d'un seul processeur par mémoire locale, de "petites" architectures SMP. Un exemple pour une telle machine ccNUMA est l'Origin 3000 ou le HP 9000 Super Dome.

– **Cluster** *Grappe*
Un cluster [9] n'est pas une machine parallèle unique au sens classique mais est constitué d'un ensemble de noeuds (ordinateurs mono-processeur ou SMP) interconnectés par un réseau local et (souvent) rapide[3]. C'est donc un réseaux d'ordinateurs en général de caractère homogène. Un des objectifs principaux d'un cluster est de fournir un environnement à image d'un système unique pour des applications destinées au cluster. Souvent, on lui associe pour-cela un système de mémoire virtuelle partagée.
Par sa nature, un cluster est composé uniquement de composants standards et possède ainsi une très bonne relation coût/performances. En plus, il est facilement extensible et devient une des architectures parallèles très répandue aujourd'hui[4].
La différence entre un cluster et un MPP décroît de plus en plus. L'existence des environnements adaptés permet aujourd'hui d'arriver à un cluster à la même puissance de calcul qu'un serveur parallèle MMP ou SMP.

Dans ce mémoire nous avons laissé volontairement de côté tout ce qui concerne le terme de *Grid* (grille ou réseau de ressources informatiques hétérogènes). La notion de *Grid* s'est seulement établie récemment en informatique et les applications spécifiques pour le calcul parallèle sur *Grid* sont encore à leurs débuts. Néanmoins, on peut voir les trois architectures présentées ici comme des noeuds de ces grilles.

[2]Voir aussi `http://box.mmm.edu/mm5/mpp.html` (voir *Benchmarking*).
[3]Par exemple Gigabit-Ethernet ou Myrinet.
[4]Voire aussi `http://www.top500.org/`

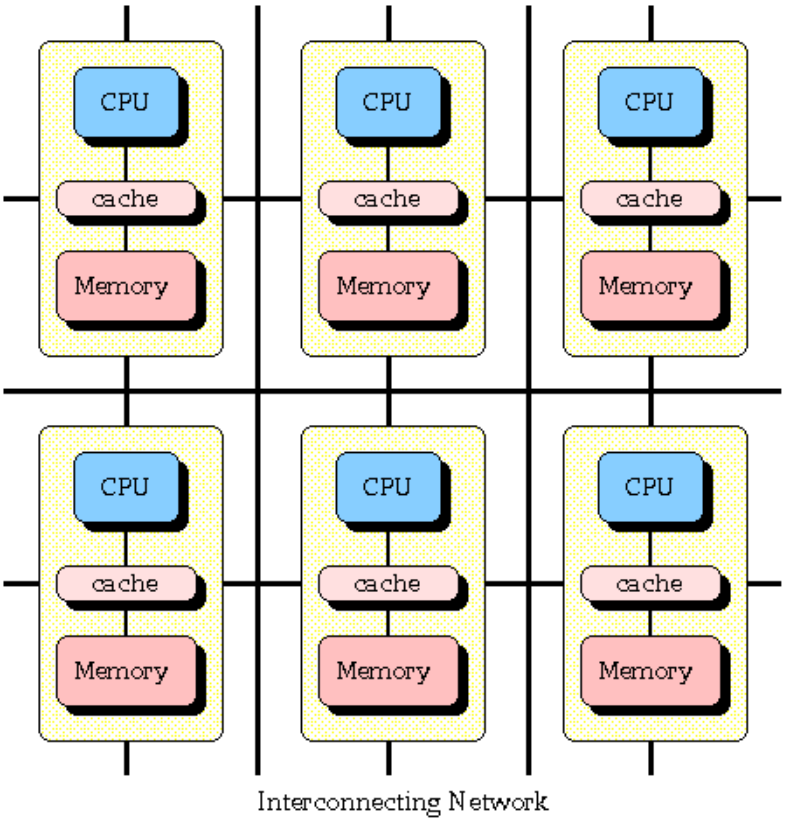


FIG. 1.3 – *Schéma d'une architecture Massively Parallel Processors (MPP)*[26].

En résumé, une grande partie des machines parallèles actuelles ne sont donc plus qu'un assemblage de matériel standard que l'on trouve dans les PC, mais reliées avec un réseau très rapide. Les machines parallèles et les réseaux d'ordinateurs sont de plus en plus semblables.
Vu l'émergence des clusters dans les laboratoires de recherche et dans les moyennes entreprises un cluster d'ordinateurs constitue l'architecture utilisée dans ce mémoire. Comme la majorité des environnements pour cluster est développée sous Linux, nous allons nous consacrer uniquement aux clusters Linux. Remarquons encore que, comme les SMP ou MPP, un cluster correspond au modèle SPMD proposé.

Vu le choix du cluster, nous allons maintenant analyser les différents problèmes qui peuvent se poser au niveau d'équilibrage de charge et d'ordonnancement de tâches sur de telles architectures avec plusieurs processeurs.

1.2 Systèmes d'équilibrage de charge

Dans le contexte des architectures parallèles présentées dans la section précédente plusieurs processeurs sont disponibles pour calculer une tâche. Une tâche est un ensemble d'opérations qu'un système ou infrastructure informatique peut effectuer afin d'accomplir l'objectif de la tâche. Des tâches peuvent avoir une multitude de caractéristiques différentes. En effet, des tâches, non nécessairement indépendantes, ne se présentent pas obligatoirement à un moment donné à notre système parallèle qui est sensé les traiter. Différentes tâches peuvent avoir des durées d'exécution assez différentes ou même la charge

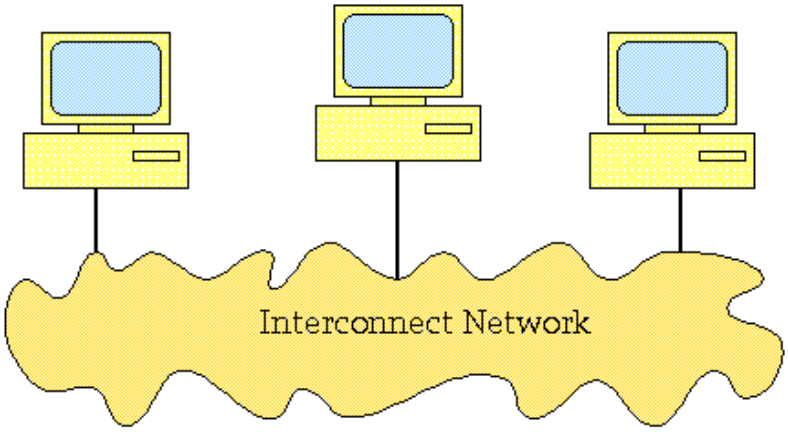


FIG. 1.4 – *Visualisation d'une architecture grappe*[26].

de calcul d'une tâche peut évoluer dynamiquement lors de son traitement. Il est donc difficile de faire un planning pour l'exécution des tâches sur les différents processeurs. En plus, par exemple dans un cluster, les tâches ne se présentent pas à un endroit unique du système. Des tâches peuvent aussi, à coté des ressources de calcul d'un processeur, avoir besoin des parties mémoires ou faire appel à des actions d'entrées/sorties. En informatique parallèle, l'existence de tâches parallèles est source d'autres problèmes comme les aspects de synchronisation, de communication et/ou de partage de données.
Toutes ces caractéristiques des tâches (que nous pouvons sûrement encore compléter) peuvent être à l'origine de déséquilibres de charge[5] sur l'ensemble du lieu de notre système. Par exemple, dans le cas d'une absence de gestion de planification des tâches sur un système parallèle à deux ordinateurs, un ordinateur A peut être submergé de tâches à traiter et les temps de réponse[6] sont fortement rallongés, tandis que son voisin B passe beaucoup temps à ne rien faire. Ces situations de déséquilibre ne sont pas rares en réalité. En effet, l'étude [22] a dégagé un principe de corrélation qui indique d'une part qu'une tâche importante en terme de consommation de ressources est successible de devenir encore plus consommatrice et d'autre part qu'un nombre élevé de tâches à un endroit est favorable à la création de nouvelles tâches à cet endroit. Selon le principe de corrélation un déséquilibre de charge éventuel a donc tendance à se fortifier.
En domaine parallèle, le ralentissement d'une tâche à cause de la charge d'un processeur provoque souvent un ralentissement de toutes les tâches qui coopèrent avec la tâche ralentie. Avec cet effet "boule de neige" un déséquilibre de charge peut influencer considérablement le temps d'exécution d'une application parallèle.
Il peut donc être avantageux d'avoir un gestionnaire qui décide où une tâche est calculée et (plus important encore si la tâche n'est pas indépendante) qui gère la coordination (communication, synchronisation, concurrence) avec d'autres tâches existantes dans le système. Un tel gestionnaire est appelé un système d'équilibrage de charge.

[5] Sous le terme charge à ce moment initial du mémoire nous classifions tous les effets que le traitement des tâches peuvent produire en terme de nombre d'instructions à exécuter par seconde, l'occupation du mémoire ou encore le nombre d'entrées et sorties.

[6] Le temps de réponse est le temps entre le lancement d'une tâche et l'accomplissement de l'objectif de la tâche par le système ou infrastructure informatique.

1.2.1 Le défi et les objectifs de l'équilibrage de charge

Les tâches d'un système multi-processeur devraient être gérées de telle manière, que la charge créée par le traitement de ces tâches soit réparti le plus équitablement possible entre les différents processeurs. La définition et l'établissement de cet équilibre constitue le défi de l'équilibrage de charge. Dans ce sens, les deux objectifs visés de l'équilibrage de charge sont :

1. minimiser le temps moyen de réponse des tâches.
2. maximiser le degré de l'utilisation des ressources.

Pour atteindre ces deux objectifs de performance le système d'équilibrage de charge [18] décide de l'ordonnancement des tâches, donc quand et où une tâche est calculée. Le système d'équilibrage de charge permet parfois aussi pour atteindre les objectifs de faire une adaptation de l'équilibrage des charges, c'est à dire de changer l'endroit d'exécution d'une tâche après sa création. Ce ré-ordonnancement des tâches s'appelle aussi migration.

Dans des architectures multi-processeurs de type SMP cette migration de tâche ne pose pas de problèmes car la mémoire allouée qui contient l'état actuel de la tâche est partagé par tous les processeurs. Donc une tâche peut être arrêtée sur un processeur et être immédiatement relancée sur un autre qui est moins chargé. En plus, la communication entre les tâches coopératives se fait sur un bas niveau.
Pour les multi-processeurs à mémoire distribuée comme les MPP le système d'équilibrage de charge doit veiller à ce que la mémoire utilisée par la tâche migrée soit aussi transportée vers le nouveau lieu d'exécution. De plus, le système d'équilibrage doit assurer que des communications éventuelles de la tâche migrée avec d'autres tâches ne soient pas interrompues.
Dans des architectures de type cluster, le débit du réseau joue de plus un rôle important pour la communication des tâches en général, mais devient aussi un facteur de coût important pour la migration d'une tâche avec sa mémoire.

1.2.2 Deux catégories principales : dynamique vs statique

L'équilibrage des charges peut se faire de deux manières différentes. Une catégorie prend en compte la dynamique d'un système informatique, l'autre par contre est statique mais plus simple à mettre en oeuvre.

Dynamique L'équilibrage dynamique prend en compte, pour l'ordonnancement des tâches, la charge actuelle des processeurs[7] dans les architectures multi-processeurs ou des noeuds en général pour le cas des clusters. S'il existe un changement de la charge pendant l'exécution d'une tâche, une migration d'une tâche vers un autre processeurs/noeuds est envisageable.
L'algorithme utilisé par un système d'équilibrage dynamique doit être capable de se renseigner sur la charge des différents processeurs ou des noeuds dans le cas d'un cluster. Ce comportement provoque un sur-coût vu l'échange de ces données statistiques. Ainsi les processeurs doivent mettre à disposition des ressources supplémentaires pour la communication. Il est important de comparer les gains de l'équilibrage dynamique avec les coûts d'administration d'un tel comportement. Ainsi, les coûts causés par une migration (arrêt d'une tâche, transfert, et relance de la tâche ; voir aussi page 14) doivent être pris

[7] On peut aussi imaginer qu'un tel système prend en compte en plus de la charge CPU, la mémoire utilisée, les opérations d'entrée et sortie. Pour des raisons didactiques, parlons ici simplement de charge de processeur en général.

en considération par des algorithmes d'équilibrage dynamique de charge, si on veut arriver à un comportement efficace du système.

Statique Le système d'équilibrage statique affecte les tâches, par des allocations uniques et définitives, aux processeurs ou noeuds d'une architecture parallèle. Il est impossible de migrer des tâches. Le gestionnaire de l'équilibrage distribue les tâches sans effectuer une demande de la charge actuelle des processeurs du système. L'algorithme utilisé ici doit donc pouvoir calculer les charges en avance, pour permettre un équilibrage de charge effectif. Donc la charge produite par une tâche doit être fournie par la tâche elle-même ou par d'autres sources. Aussi, les différentes capacités des processeurs ou noeuds de réseau doivent être connues.

L'équilibrage statique est plus simple que la manière dynamique. Mais l'équilibrage dynamique peut se montrer plus performant dans le cas où les conditions de l'équilibrage statique ne sont pas établis. En effet, souvent il est difficile voire impossible de connaître les charges qu'une tâche produit en avance. Par exemple, même si on sait qu'une tâche dans les précédentes exécutions s'est toujours terminée sans avoir utilisé beaucoup de ressources, alors pour la prochaine exécution rien ne lui interdit, d'être très gourmande en ressources. Ce problème se pose spécialement en domaine parallèle, où nous avons des variables externes comme la charge du réseau où encore l'attente de résultat d'autres tâches qui rendent difficile l'ordonnancement effectif des tâches. Aussi l'arrivée continue de nouvelles tâches rend difficile la gestion d'ordonnancement par un équilibrage statique. En effet, la non-connaissance des charges futures pose problème dans un ordonnancement statique efficace. Dans des environnements très dynamiques, il est même possible qu'un équilibrage statique crée des déséquilibres plus importants que l'équilibrage produit par une distribution au hasard des tâches. Donc le besoin de pouvoir adapter les estimations initiales est justifié.

Vu que le noeud d'un cluster peut être constitué lui-même d'une architecture multiprocesseurs, on peut imaginer qu'un tel noeud peut avoir son propre système d'équilibrage de charge local non nécessairement de la même catégorie pour gérer la charge à ce sous-niveau. Dans la section suivante nous allons voir d'autres stratégies et structures qu'un système d'équilibrage de charge peut prendre.

1.2.3 Stratégies de l'équilibrage

Nous traitons maintenant trois questions importantes [49] à propos de l'équilibrage des charges. Ces questions sont 1) le lieu où la décision de l'équilibrage des charges est prise, 2) quelles informations sont utilisées pour faire cette décision et 3) qui prend la décision de l'équilibrage des charges.

A) L'équilibrage central ou distribué des charges

Pour le lieu de la décision d'équilibrage nous distinguons entre un équilibrage central et distribué.

Central Dans une gestion centralisée un processus central prend en charge la totalité des ordonnancements et/ou ré-ordonnancements des tâches. L'avantage d'un point central est qu'il constitue un lieu unique d'entrée pour de nouvelles tâches. Le système central a une connaissance totale du système et les tâches peuvent alors être distribuées d'une manière équilibrée. A partir d'une certaine taille du système parallèle ou d'un certain

nombre de tâches le passage par un point unique peut devenir contraignant. Un exemple pour le management central est un serveur web.

Distribué Le point central, souvent contraignent dans des systèmes de plus grande taille, est supprimé dans le management distribué. Ici chaque partie du système[8] prend en charge une partie précise de l'administration. Par exemple, dans le cas des clusters, chaque noeud héberge une partie du système d'équilibrage. Pour arriver à des prises de décision cohérentes et efficaces au sein du système distribué, les différentes parties possèdent un moyen de communication. Ainsi, les noeuds de notre cluster vont communiquer entre eux pour établir l'ordonnancement et le ré-ordonnancement global des tâches. Mais, la distribution des charges ne s'effectue pas nécessairement selon un algorithme identique et connu par toutes les parties du système. On pourrait aussi imaginer un système coopératif avec des parties qui ont des stratégies ou des algorithmes d'équilibrage différents mais qui communiquent par exemple via des interfaces communes. En effet, pour le cas de notre cluster, un système d'équilibrage central pourrait faire un premier équilibrage global entre les noeuds du cluster et communiquerait avec des systèmes d'équilibrage distribué installés sur chaque noeud pour effectuer un équilibrage local plus détaillé.
L'avantage d'un tel système distribué est sa tolérance aux pannes et son extensibilité. Par contre, les solutions distribuées ont un sur-coût dû aux communications supplémentaires.

B) Stratégies globales ou locales

Des polices ou politiques déterminent quelles informations sont utilisées pour prendre une décision d'équilibrage des charges.
Dans des polices globales le management d'équilibrage utilise toutes les informations disponibles du système. Dans de grands systèmes, l'échange d'informations provoque beaucoup de communications et de synchronisations supplémentaires.
Pour réduire les échanges d'informations les polices locales regroupent et résument les informations d'une partie du système[9]. Ainsi, seulement ces résumés d'informations sont échangés au niveau global du système.
Aussi avec des restrictions sur les communications le besoin permanent des ressources pour l'administration du système d'équilibrage de charges est limité. Par exemple, il est possible d'imaginer un cluster où chaque noeud communique uniquement avec un nombre fixe d'autres noeuds voisins. Le système est ainsi totalement extensible.

Les informations qui devraient être prises en considération par le gestionnaire pour un cluster sont par exemple :

- la capacité disponible sur chaque noeud (CPU, taille de la mémoire),
- la charge actuelle du noeud,
- la capacité requise par chaque tâche,
- la capacité du réseau,
- les plans de communication et d'interdépendance (échange de données, synchronisation, ...) entre les tâches.

Cependant, la mesure de toutes ces informations n'est pas facile. Une solution est que l'utilisateur ou le programmeur d'une tâche fournisse des informations nécessaires. Aussi, des systèmes d'exploitation comme Unix proposent une série de commandes qui permettent au gestionnaire d'interroger la capacité ou la charge d'un système.

[8] Partie du système désigne ici un sous-système du système qui est capable de gérer l'équilibrage de charge lui-même.

[9] Partie du système désigne ici un ensemble de processeurs ou un noeud d'un cluster.

C) Stratégies actives ou passives

Pour la responsabilité de l'équilibrage nous pouvons distinguer entre des algorithmes actifs à l'initiative de l'envoyeur et passifs à l'initiative du destinataire :

- **Actif** Nous parlons d'un algorithme actif, si des voisins négocient pour l'ordonnancement des tâches. Par exemple le noeud N demande la charge d'un noeud M. M donne sa charge actuelle et demande en retour également la charge de N. Selon le résultat d'une comparaison des charges, des tâches sont re-ordonnancées. Nous parlons ici de l'initiative de l'envoyeur car cette technique est souvent utilisée pour libérer un processeur ou noeud sur-chargé.

- **Passif** Ici il n'existe pas de discussion entre les noeuds. Par exemple le noeud N remarque que sa charge est trop faible. Selon le principe appelé *workstealing* [6] il demande alors du "travail" au noeud M. A l'inverse de l'actif c'est ici le destinataire qui vole du travail pour éviter une situation d'inoccupation.
  Un exemple pour un équilibrage statique central est de s'imaginer qu'il existe une sorte de file au point central qui contient les tâches qui doivent être traitées. Le "demandeur de travail" contacte alors le gestionnaire de cette file qui va lui attribuer une tâche selon la politique de gestion de la file.

1.2.4 La migration des tâches

Comme introduit au-dessus, avec la migration il est possible de changer le lieu de l'exécution d'une tâche déjà créée, pendant son exécution.
Si la migration d'une tâche est souhaitée de nouveaux problèmes apparaissent. L'algorithme d'équilibrage des charges doit disposer de critères pour choisir une tâche appropriée et sa destination. Puis, la tâche retenue doit être arrêtée sur le processeur actuel. Pour que la tâche puisse continuer son exécution sur la destination toutes les références objets et variables doivent être identifiées. Ce mécanisme est parfois appelé *checkpointing*.
Maintenant, la tâche peut être transférée vers le système destinataire. Pour les objets et variables il existe deux possibilités :

1. Tous les objets et variables sont également transférés. La tâche migrée doit être la seule qui utilise ces objets et variables.

2. Les objets et variables restent sur le système initial. La tâche transférée doit accéder aux objets et variables, maintenant distants, via des références. Dans un système réseau ces appels à distance provoquent une charge supplémentaire pour le réseau. Aussi le système initial a des charges supplémentaires dues aux appels distants.

Finalement, pour chaque décision de migrer une tâche le gestionnaire doit respecter les deux critères suivants :

- La migration d'une tâche va augmenter la charge de la destination. Pour éviter une oscillation de déplacement de tâches, le gestionnaire doit s'assurer que la migration ne provoque pas une charge sur la destination qui est plus grande que la charge de l'hôte sur lequel la tâche réside actuellement.
- La mesure du succès du l'équilibrage de charge dynamique est la différence du temps d'exécution obtenu par l'algorithme d'équilibrage dynamique. Pour cela, le gestionnaire doit uniquement migrer des tâches dont le gain potentiel en temps d'exécution est plus élevé que le coût de migration de la tâche considérée.

1.3 Modèles de programmation parallèle

En général, la programmation parallèle essaye d'utiliser la concurrence dans un programme, afin de réduire le temps de réponse pour résoudre un problème ou d'augmenter la taille des problèmes qui peuvent être résolus. La programmation parallèle est donc plus performant pour solutionner un problème posé.

Les modèles de programmation parallèle dépendent fortement des architectures parallèles sous-jascentes. Comme déjà énoncé ci-dessus nous ne pouvons pas programmer une machine avec une mémoire partagée de la même manière qu'une machine avec une mémoire distribuée.
Ainsi pour la programmation parallèle, nous ne pouvons pas appliquer les mêmes principes qu'en programmation séquentielle. En programmation parallèle le programmeur doit décider s'il applique le parallélisme de données, c'est à dire si on fait la même chose sur les données différentes, ou s'il opte pour un parallélisme de contrôle sur l'ordonnancement des tâches ou encore s'il applique les deux parallélismes simultanément. Ainsi, nous pouvons définir différents grains de parallélisme qui peuvent aller du niveau du programme jusqu'au niveau des bits. D'autres problèmes sont dûs à des communications et synchronisations de tâches parallèles. Dés lors, avec des délais introduits par des réseaux et l'absence du temps global (voir figure 1.5) dans un système distribué, l'exécution d'un programme parallèle n'est pas totalement déterministe. En effet, par exemple suite à la charge d'un réseau ou encore à la différence des distances entre les lieux d'exécution des tâches parallèles, les communications ne peuvent pas garantir l'ordre causal des messages, ce qui pose entre autre un problème pour le débuggage et la vérification sémantique[10] des programmes parallèles.

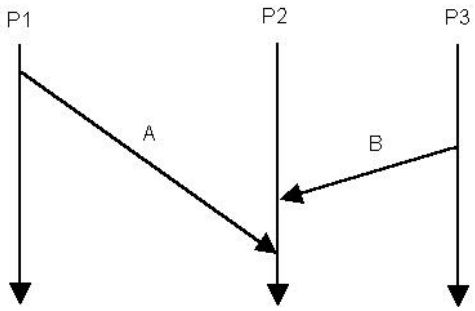


FIG. 1.5 – *Visualisation de l'absence de temps global externe (avec différence des distances et différents délais de réseau). "Un message A est plus ancien qu'un message B si A et B arrivent sur le même processeurs P et si A arrive avant B indépendant de leur ordre d'émission par rapport à un temps global externe."*

Dans cette partie nous allons d'abord voir des mesures de l'efficacité d'un programme parallèle par rapport à un programme séquentiel. Un exemple des difficultés de transformation d'un programme séquentiel en programme parallèle est donné au chapitre 5. Puis, nous allons voir trois modèles de programmation parallèle ([9], [34] et [13]) utilisés dans ce mémoire.

[10] C'est dire pas de deadlock, pas d'accès concurrents au mêmes données, ...

1.3.1 Speedup et loi d'Amdahl

Une raison pour l'introduction de la programmation parallèles est le gain de temps d'exécution par rapport au programme séquentiel. Il découle donc, très naturellement, qu'on évalue les programmes parallèle par rapport à cette diminution du temps de réponse.

Définissons d'abord la notion de ***speedup*** ou accélération des programmes parallèles. Soit T_1 le temps nécessaire à un programme pour résoudre le problème A sur une architecture séquentielle et soit T_p le temps nécessaire à un programme pour résoudre le même problème A sur une architecture parallèle contenant p processeurs, alors le *speedup* est le rapport :

$$S(p) = \frac{T_1}{T_p} \tag{1.1}$$

Cette définition n'est pas très précise. Par exemple, pour le programme qui donne le temps T_1 nous pouvons prendre le programme parallèle configuré afin qu'il s'exécute sur un seul processeur, un programme séquentiel utilisant le même algorithme parallèle ou encore le programme séquentiel le plus rapide résolvant le même problème que le programme parallèle. Ainsi, pour pouvoir comparer des *speedups* de différents programmes on doit spécifier le programme séquentiel de référence ainsi que les architectures à la base des mesures.

Un autre concept limite l'accélération des programmes parallèles. En effet, le *speedup* d'un programme sera limitée par le pourcentage de code intrinsèquement séquentiel qu'il contient. Ce principe est connu sous la **loi d'Amdahl** et se base sur le fait que chaque programme parallèle contient une partie séquentielle. En fait, le temps d'exécution T_1 d'un programme séquentiel peut être décomposé en deux temps :

1. T_s consacré à l'exécution de la partie intrinsèquement séquentielle
2. $T_{//}$ consacré à l'exécution de la partie parallélisable

$$T_1 = T_s + T_{//} \tag{1.2}$$

Seul $T_{//}$ peut être diminué par la parallélisation. Dans le cas idéal, on obtiendra au mieux un temps $T_{//p}$ pour la partie parallélisée. Ainsi, on obtient la loi d'Amdahl :

$$T_p \geq T_s + T_{//p} \tag{1.3}$$

Par exemple, si on fixe le temps T_1 à une unité et si on suppose qu'on peut parallélisé 75% d'un programme séquentiel (donc $T_s = 1/4$), alors avec la loi d'Amdhal le *speedup* est limité à 4 :

$$\begin{aligned} S(p) &= \frac{T_1}{T_p} && avec\ (1.1) \\ &\leq \frac{1}{T_s + T_{//p}} && avec\ T_1 = 1\ \ et\ \ la\ loi\ d'Amdahl\ (1.3) \\ &\leq 4 && avec\ T_s = \frac{1}{4}\ \ et\ \ T_{//p} > 0 \end{aligned}$$

1.3.2 Paradigme de programmation *multi-thread*

La programmation parallèle la plus simple consiste à avoir plusieurs *threads* ou flots d'instruction qui s'exécutent en manipulant des données stockées dans une mémoire commune. Elle est donc naturellement le type de programmation le plus employé sur les

machines SMP. La programmation peut se faire avec des outils spécialisés comme les bibliothèques implantant la norme POSIX [8]. Elle peut aussi utiliser des langages connus qui intègrent les *threads* dans leur sémantique, comme Java.
La difficulté de cette programmation est de garantir la sémantique correcte du programme en synchronisant les accès à la mémoire entre calculs se déroulant de manière concurrente. Pour obtenir des performances, il suffit d'avoir toujours suffisamment de calculs concurrents à faire pour occuper les différents processeurs. On se base ici sur le principe du *multi-threding* où chaque processeur exécute en alternance plusieurs *threads*. Après un certain temps de calcul ou quand un *thread* passe dans un état d'attente le processeur bascule alors à un autre *thread* qui est en attente d'exécution.

L'avantage du *multi-threading* est aussi utilisable sur des architectures sans mémoire partagée où la communication et des caractéristiques réseau comme le délai jouent des rôles plus importants. Avec le principe de *multi-threading* il est alors facile de combler par exemple l'attente d'une synchronisation ou d'une communication, comme l'attente d'une donnée d'un autre *thread*, par d'autres calculs (principe *asynchrone*) ou de remplir le temps de la communication en terme de délai du réseau par des calculs (principe *non-blocking*) en implémentant des *threads* qui gèrent les protocoles de communication. Cette utilisation du *multi-threading* est aussi connu dans la littérature sous le nom de parallélisme de tâche (ou parallélisme fonctionnel) et consiste donc de décomposer un programme en plusieurs tâches qui peuvent alors être exécutées simultanément.

Un désavantage du *multi-threading* est le sur-coût qui est induit à cause du basculement entre *threads*. En effet, on doit enregistrer l'environnement d'exécution du *thread* qu'on veut mettre en attente et chargé les registres du processeur avec le contexte d'exécution du nouveau *thread*. Cependant, il existe des architectures avec des jeux de registres qui permettent de garder plusieurs environnements d'exécution de *threads* à la fois. Une autre solution pour diminuer le sur-coût du passage entre les *threads* sur un processeur est de créer un environnement qui implémente un passage entre des *threads* qui est plus performant et spécialisé à une classe de problème. Ainsi, l'environnement cache le *multi-threading* devant le système exploitation qui, lui, doit fournir de la gestion des *threads* beaucoup plus complet et dès lors plus lourds. Les *threads* pris en charge par un tel environnement sont souvent appelés *user-level threads*.

1.3.3 Le paradigme *message-passing*

La programmation d'une machine à mémoire distribuée impose plusieurs contraintes. Il faut exprimer explicitement quelles vont être les données transmises et à quels moments les transmissions doivent se faire. La façon standard de programmer est d'utiliser une bibliothèque d'échanges de messages comme MPI[36] ou PVM [42], qui permet de s'abstraire de la réalité physique de la machine ou du réseau, et de se concentrer sur les échanges de données. En effet, ces bibliothèques sont implémentées pour la plupart des architectures parallèles et permettent donc d'écrire en principe des programmes portables et indépendants des machines.

Dans le paradigme de programmation *message-passing* [30], les programmeurs voient leurs programmes comme une collection de processus avec une mémoire privée. En général, en *message-passing* on travaille avec autant de processus que de processeurs. Ainsi, l'environnement de *message-passing* peut affecter un processus à un processeur et l'échange de message entre processus revient à un échange de message entre processeurs. C'est aussi l'environnement qui spécifie un identificateur pour chaque processus.

La communication entre processus et l'échange de données se fait en général avec deux primitives fournies par les bibliothèques d'échange de messages. La première est la primitive d'envoi `send`. Sa syntaxe est toujours proche de la suivante [20] :

`send(buffer, taille, destinataire, type)`

- `buffer` est une référence en mémoire qui indique où trouver le premier octet du message à envoyer
- `taille` est un entier indiquant la taille du message
- `destinataire` est l'identification du processus destinataire
- `type` est le type du message, souvent un entier. Il permet de classer les messages du point de vue de l'application.

La deuxième, le complémentaire de `send`, est la primitive de réception `recv`. Sa syntaxe est proche de [20] :

`recv(buffer, taille, expediteur, type)`

- `buffer` est une référence en mémoire qui indque où devra être déposé le premier octet du message
- `taille` est un entier indiquant la taille du message
- `expediteur` est l'identification du processus l'expéditeur
- `type` est le type du message attendu.

Un message peut seulement être envoyé si l'expéditeur appelle explicitement la primitive `send` et le destinataire sa primitive `recv`.

Ces deux primitives sont en général disponibles en plusieurs modes de communication. Citons ici quatre modes de comportement possibles dans une communication synchrone, asynchrone, bloquante et non-bloquante [41]. Par exemple, les primitives par défaut en MPI sont asynchrones et bloquantes. **Bloquant** veut dire que l'appel à `send` ou `recv` retourne uniquement si le buffer associé est libéré, c'est à dire pour le `send` que l'outil de communication a "copié" le message pour l'envoi et pour le `recv` que le message reçu se trouvent dans le buffer (voir figure 1.6). Et **asynchrone** exprime ici le fait que `send` ne sait pas si `recv` a reçu le message ou pas.
En général, les bibliothèques fournissent aussi des primitives pour le non-bloquant et le synchrone. Le principe non-bloquant est introduit pour augmenter les performances du système. En effet, les primitives **non-bloquantes** retournent directement et rendent ainsi possible de faire des calculs pendant la communication. Dans le cas du `send` il y un risque d'écraser le buffer avant que le message ne soit envoyé. Le `recv` est complété par un appel de `wait` qui est bloquant jusqu'à ce que le message soit disponible dans le buffer qui est spécifié par `recv`. **Synchrone** exprime ici le fait que l'expéditeur est sûre que le message est arrivé chez le destinataire, c'est à dire que le `send` retourne uniquement si le `recv` correspondant est terminé.

En plus des deux primitives principales, les modèles de *message-passing* fournissent en général aussi des moyens de communication plus globaux ou plus performants adaptés à des cas particuliers. Ce sont les communications collectives. Par exemple, pour envoyer un message à tous les processus il existe un appel `broadcast` (diffusion général) et pour récolter des résultats d'un calcul de tous les processus chez un processus un appel `reduce`. Néanmoins, les communications sur des machines à mémoire distribuée sont toujours assez coûteuses en temps par rapport à leur puissance de calcul. Cela induit que la programmation parallèle devient plus complexe : il faut essayer de minimiser les communications, dans le même temps, que de partager les données et les activités pour maintenir les processeurs occupés.
Sur les machines à mémoire virtuellement partagée ccNUMA, la programmation est plus

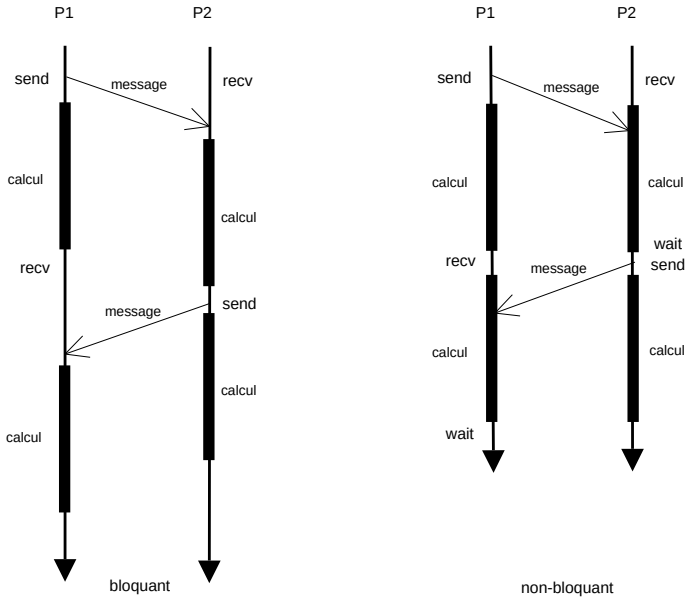


FIG. 1.6 – *Différence entre des modes de synchronisation bloquante et non-bloquant*

aisée (comme sur une machine SMP). Mais, pour obtenir une application efficace, il faut résoudre les mêmes problèmes que sur les architectures à mémoire distribuée car les différences d'ordre de grandeur entre puissance de calcul et temps de transmission des données n'ont pas changé.

Le paradigme de *message-passing* s'applique bien au modèle **SPMD** (Single Program Multiple Data). En effet, les bibliothèques d'échange de messages mettent à disposition des moyens pour identifier les différents processus. Ainsi chaque processus peut effectuer en parallèle un calcul identique sur une partie des données. Aussi, par exemple dans des problèmes de multiplication de matrices, il est possible d'échanger avec le *message-passing* des résultats intermédiaires pour la suite des calculs (voir figure 1.7). Un problème dû à la distribution des calculs en SPMD est de savoir quand tous les calculs sont terminés. Le difficulté est de définir quand et comment une application peut se terminer.

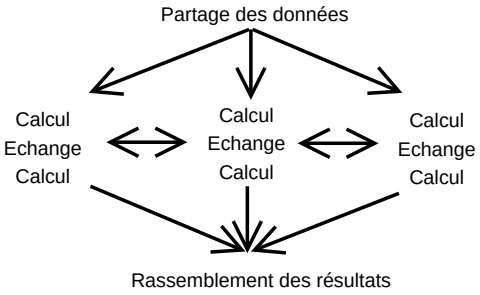


FIG. 1.7 – *Structure générale d'un programme SPMD*

Une technique complémentaire au SPMD, est d'attribuer la partie séquentielle d'une application à un processus maître, qui va alors s'occuper de distribuer les données aux processus, appelés esclaves, pour le calcul en parallèle et qui rassemble finalement les résultats partiels produits par les esclaves (figure 1.8). Ce modèle **maître-esclave** permet

aussi de résoudre le problème de terminaison grâces aux capacités de gestion du maître. Le désavantage du modèle est que le maître peut vite devenir un élément *bottleneck* de l'application parallèle si les calculs parallèles sont d'une taille réduite.

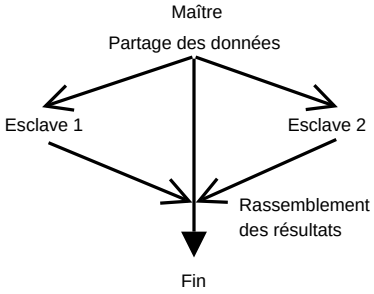


FIG. 1.8 – *Structure générale d'un programme Maître-Esclave*

Dans le cas, où les processus parallèles échangent des résultats intermédiaires, on applique souvent une **approche itérative**. Une itération consiste alors à calculer des résultats intermédiaires. Puis, d'effectuer une synchronisation qui garantit que tous les résultats intermédiaires sont disponibles pour effectuer finalement les échanges (figure 1.9). Les barrières de synchronisation sont fournies aussi par les bibliothèques du *message passing*.

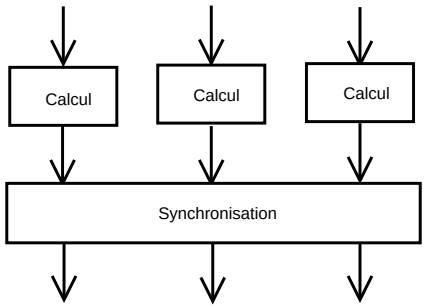


FIG. 1.9 – *Une itération de l'approche itérative*

1.3.4 Le modèle de programmation *message-driven*

Le modèle de programmation *message-driven* essaye d'améliorer les performances du *message-passing* en utilisant des techniques de l'orienté objet et du *multi-threading*.
En effet dans le *message-passing* traditionnel, une communication entre deux processus oblige l'expéditeur et le destinataire d'appeler explicitement et respectivement leurs primitives d'envoi ou de réception. Par défaut, ces deux primitives sont bloquantes et spécialement un appel de `recv` bloque jusqu'au moment où le message est arrivé. Cette synchronisation entre l'arrivée du message et le `recv` est source de temps d'inoccupation du processeur et n'est peut-être pas toujours nécessaire. Il existe des techniques de programmation qui essayent d'avancer les `send` et de retarder les `recv` pour diminuer le temps d'attente du processeur. Ces techniques sont cependant limitées vu les difficultés en cas de communication plus complexes entre processus. Les appels de primitives non-bloquantes en *message-passing* pouvaient être une solution à ce problème, or on doit toujours se référer à des appels comme `wait` qui bloquent alors à leur tour jusqu'à l'arrivé du message. Aussi il y a des travaux comme [15] qui montrent, que même si théoriquement des appels

non-bloquants en *message-passing* sont possibles, il existe des implémentations de MPI qui ne peuvent pas les fournir à cause des protocoles de communication sous-jascents de certaines architectures.

Le modèle de *message-driven* veut combler ce défi. Le principe du *message-driven* est que le calcul sur un processeur est provoquer par l'arriver d'un message. Ce calcul génère à son tour des nouveaux messages qui vont donc provoquer d'autres calculs, si possible distants pour distribuer et paralléliser le calcul. Le modèle *message-driven* est aussi souvent caractérisé comme un modèle orienté-message.
Un calcul est donc seulement effectué si un message arrive. Pour réagir aux arrivées de messages, le modèle de *message-driven* utilise des sortes d'objets, appelé objets de *message-driven*. Chaque objet *message-driven* doit fournir des méthodes d'entrée qui peuvent alors être appelées via des messages par d'autres objets *message-driven*.
Ces méthodes d'entrée ne retournent aucun résultat (méthodes *void*) aux appelants de la méthode. Ce procédé asynchrone (voir aussi en-dessous) permet que l'envoi d'un message, c'est à dire l'appel d'une méthode d'entrée, ne bloque pas l'expéditeur donc l'appelant de la méthode entrée.
Si un message arrive maintenant pour une telle méthode d'entrée d'un objet *message-driven* c'est le environnement de *message-driven* qui va s'occuper de le réveiller et d'appeler la méthode d'entrée qui correspond au message. Ainsi, le modèle *message-driven* à supprimer l'attente sur un message, car la primitive `recv` ne doit plus être appelée explicitement dans le code de l'objet.
Le principe consiste alors de placer différents objets *message-driven* sur un processeur. Dès qu'un objet *message-driven* a terminé l'exécution de sa méthode il passe dans un état inactive et le processeur peut alors passer à l'exécution d'une autre méthode entrée d'un autre objet *message-driven* pour lequel un message est en attendant. Ainsi, on peut limiter l'inoccupation du processeur pourvu qu'il y a toujours en message qui correspond à une méthode d'entrée d'un objet *message-driven* du processeur.
Remarquons, que ce passage entre objet ressemble fortement au *multi-threading*, c'est à dire des *threads* qui sont exécutés à tour de rôle. En effet, une possibilité d'implémenter les objets *message-driven* est de créer des sortes de *thread*. Spécialement, pour transformer un processus *message-passing* en objet de *message-driven*, on l'implémente souvent comme un *user-level thread*. Ainsi, si un *user-level thread* qui représente un objet *message-passing* est bloqué par l'attente d'un message, le processeur peut passer à un autre *user-level thread* et rétablit ainsi le principe du *non-blocking* d'un processeur en *message-driven*. L'utilisation des *user-level thread* s'explique entre autre par la volonté de diminuer les coûts de passage entre *threads* sur un processeur.

Les objets de *message-driven* ne sont pas liés à un processeur spécifique. Ainsi, il est facile d'effectuer des migrations d'objets d'un processeur à un autre pour mieux équilibrer les charges et de réduire les temps d'inoccupation des processeurs. Vu que les messages envoyés via des appels de méthode sont destinés à une méthode d'entrée d'un objet *message-driven* et non plus comme en *message-passing* à un processus d'un processeur, les messages, aussi vus comme des objets par l'environnement *message-driven*, peuvent être redirigés par ce dernier vers le nouveau emplacement de l'objet. Pour réaliser cette transparence de localisation, on applique souvent un système de *forwarding* pour faire suivre les messages aux objets migrés. Des environnement de *message-driven* peuvent alors fournir de l'équilibrage dynamique de charge.
Indépendamment du ou des système(s) d'exploitation de l'architecture parallèle sous-jascente, c'est donc l'environnement *message-driven* qui fait la gestion dynamique des objets ou dans le cas du *multi-threading* des *user-level threads*. Vu le rôle étendu de ges-

tion, notre environnement est aussi parfois appelé “support d’exécution”.

Pour le lecteur qui a des connaissances en *middelware* on peut comparer le fonctionnement du *message-driven* avec des invocations à distance de méthode du type “ONEWAY”, donc un appel asynchrone sans retour de réponse, de CORBA ou RMI. Le *message-driven* peut être vu comme un *Asynchronous Remote Methode Invocation*.
Pour le lecteur peu familiarisé avec ces concepts remarquons encore comment on peut remplacer un appel synchrone classique de CORBA ou RMI, c’est à dire un appel qui retourne aussi son résultat, avec deux appels du mode asynchrone. Supposons l’objet ou le *thread* A qui a besoin des données de l’objet ou *thread* B. Dans un premier temps A fait un appel à une méthode de B en spécifiant en plus une méthode pour la destination de la réponse (ici A). Comme c’est un appel asynchrone A peut exécuter après d’autres instructions ou devenir inactif. Dans un deuxième temps, l’environnement *message-driven* récupère le message envoyé par A, et ordonne à l’objet B, peut-être sur un processeurs distant, de traiter le message. B exécute la requête et envoie la réponse dans appel de retour (ici A) spécifié dans le message de A. De nouveau, B exécute d’autres instructions ou libère le processeur. Finalement, l’environnement recueille le message pour A qui contient la réponse de B et ordonne à A de le traiter (figure 1.10).

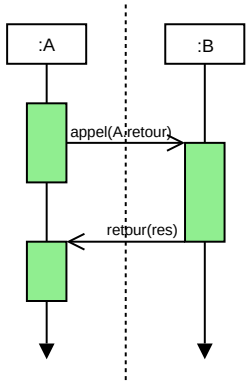


FIG. 1.10 – *Retour d’un résultat d’un appel avec deux appelles asynchrones*

En général, le *message-driven* utilise donc l’asynchronisme des flots de calculs pour recouvrir les temps d’attentes des communications et de gagner ainsi en efficacité.
Comme déjà dit, avec le *multi-thraeding* le *message-driven* n’est pas limité au paradigme orienté-objet. Comme nous allons le voir il existe des environnements qui fournissent des avantages du *message-driven* pour des programmes écrits en C. Notamment, l’environnement AMPI que nous allons analysé plus précisément constitue un tel environnement pour des programmes classique MPI écrits en C.

1.3.5 Autres approches

Pour un souci de complétude, nous allons brièvement compléter notre explication. Mis à part les trois approches présentées au-dessus, il existe des approches qui combinent deux ou trois des modèles présentés. Par exemple, Athapascan-0[2] combine les fonctionnalités d’une bibliothèque d’échange de message comme MPI et d’une bibliothèque de contrôle de flots multiple comme les POSIX *threads*.
Tout une autre approche est présenté dans le modèle BSP (Bulk-Synchronous Parallel) [4]. Dans un souci d’extensibilité et de portabilité ce modèle donne un cadre abstrait pour le design, l’analyse et la programmation de système parallèle.
Aussi des approches issues, comme déjà cité au-dessus, du domaine distribué ou *midelware*

comme les RPC (*Remote Procedure Call*) sont fort utilisés pour créer des programmes parallèles. Ou plus récent encore, la notion de "service" dans les calculs pour *Grid* gagne de l'importance.
Les modèles discutés jusqu'a présent ont une caractéristique commune qui est que le programmeur doit programmer l'aspect parallèle explicitement. Il existe cependant des outils qui tentent de paralléliser automatiquement des applications. On trouve par exemple le langage HPF[23]. Cette approche de parallélisation implicite soufre néanmoins de quelque difficultés, notamment au niveau de l'efficacité du code produit. D'autres approches, comme OpenMP[40] ou Cilk[17] est de donner des directives au compilateur pour lui aider à prendre ses décisions de parallélisation.
Pour des classes d'application bien particulières, notamment dans le domaine du calcul numérique, le problème consiste plutôt à exécuter de manière parallèle des opérations bien connues sur des données fournies par l'utilisateur. Par exemple, il peut s'agir d'effectuer le plus rapidement possible une opération matricielle sur un ensemble de processeurs. De telles fonctionnalités spécialisé sont par exemple fournies par des bibliothèque de haut niveau spécialisées comme Blas[5] ou ScaLAPACK[48].

A partir de cette dernière section introductive nous pouvons passer dans un état d'art plus complet des environnements parallèles. Le survol de ses environnements et leurs capacités d'équilibrage de charge sera présenté dans le chapitre suivant.

Chapitre 2

Description de l'état de l'art

Ce deuxième chapitre a pour objectif de donner une vue sur l'état de l'art du domaine cible du mémoire, à savoir les environnements de calcul parallèle avec des caractéristiques d'équilibrage de charge. Pour faire la relation avec le domaine du parallélisme et pour pouvoir mieux situer les différents environnements, nous citerons aussi quelques langages traditionnels (par exemple MPI) qui sont à la base des langages *high-level* auxquels nous nous intéressons (par exemple AMPI).
Dans le domaine de programmation parallèle, il existe des centaines de projets et de développements plus ou moins importants. Le portail IEEE du *parallel computing*[1] contient une liste assez exhaustive des projets de ce domaine. En plus, la plupart des projets parallèles datent de la deuxième partie des années nonante et sont parfois remplacés ou complétés par des recherches sur des applications *grid*. Vu la multitude de projet et la difficulté de caractériser un état actuel, ce document n'a pas comme objectif d'être complet dans sa description du domaine et nous allons nous limiter à une synthèse de quelques environnements "classiques" de la littérature.

Nous allons donc à partir des exigences d'un système parallèle décrire certains environnements parallèles qui sont en relation avec le thème du mémoire en les classant dans différents modèles identifiés dans la deuxième section.

2.1 Les exigences des systèmes parallèles visés

Beaucoup d'environnements traditionnels de programmation parallèle (MPI, PVM, ...) sont des langages séquentiels augmentés de quelques appels de systèmes. Ils fournissent des commandes qui permettent de paralléliser un code séquentiel.
Par contre, beaucoup d'autres tâches de programmation parallèle sont encore laissées dans les mains du programmeur de l'application. Citons par exemple : la communication et synchronisation entre processus, la répartition et distribution des données sur une architecture parallèle, l'allocation des processus à des processeurs et l'entrée/sortie des structures de données. Nous caractérisons de tels systèmes comme *low-level*.

Par rapport à ces défauts, nous spécifions qu'un système parallèle, devrait respecter les exigences suivantes [29] :

– **Généralité :** Le système ne doit ni se limiter à quelques domaines d'applications, ni à quelques paradigmes de programmation parallèle. Son langage doit être unifor-

[1] Voir `http://dsonline.computer.org/parallel/parallel-parascope.htm`.

mément utilisable par des constructeurs d'architecture, des implémentations individuelles ou par des utilisateurs finals.
- ***High-level*** **:** Un système *high-level* doit, en plus de sa généralité, s'occuper, dans la mesure du possible, de l'ordonnancement et de l'allocation des tâches.
  Remarquons que la tâche de la décomposition de l'application est encore laissée au programmeur, vu la complexité et les problèmes existants, surtout au niveau de l'efficacité du code produit, d'une telle automatisation. Par contre, un système *high-level* se caractérise par un ordonnancement et une gestion des ressources efficaces. Par exemple, un système high-level s'occupe de l'équilibrage de charge, des optimisations pour les communications et les synchronisations, ...
- **Efficacité :** Le système doit permettre une conception et implémentation efficaces. Nous prenons notamment en charge les deux notions d'efficacité suivantes :
  - ***La latence de la communication.*** Comme l'accès aux données distantes est plus coûteuse que l'accès local[2], le langage du système doit permettre d'écrire des programmes qui tolèrent et cachent ce coût de temps.
  - ***Efficacité prévisible.*** Suivant l'exigence de généralité, il doit être possible de prédire l'efficacité du système indépendamment de l'architecture utilisée. Il s'agit ici de construire un modèle de coût bien défini, avec lequel seulement il est possible de créer des applications qui garantissent des efficacités portables.
- **Expressivité :** Un système de programmation parallèle doit avoir la capacité d'exprimer la décomposition des tâches (y compris la création dynamique des tâches) et d'exprimer le partage d'information entre les différentes tâches d'une manière expressive.
- **Modularité, Scalabilité et Réutilisabilité :** La modularité est importante dans un environnement varié et la scalablité est introduite pour un désir d'extensibilité facile du système qui peu passer d'un processeur à plusieurs centaines de processeurs. Aussi, les sous-routines (éventuellement séquentielles) existantes doivent être réutilisable dans le système.

Cette grille d'analyse est utile pour effectuer le survol du domaine dans les sections suivantes. Spécialement, nous allons comparer chaque environnement décrit avec les différentes exigences présentées ici.

2.2 La classification utilisée pour le survol sur les environnements de programmation parallèles

Notre essai de parcourt se base sur un ensemble de modèles identifiés par les auteurs [43], afin de pouvoir mieux situer les différents environnements. De l'ensemble de modèles identifiés par ces auteurs nous reprenons quatre classes de modèles légèrement adaptés à notre propos. Ces modèles se basent sur :
- **le compilateur**
- **la coordination par messages**
- **le concept de l'objet**
- **le concept de tâche**

L'approche de classification n'est cependant pas facile vu que quelques environnements peuvent être classés dans différentes classes. Par exemple, l'environnement Charm++

[2]Le coût pour l'accès distant est la somme du coût de l'ouverture de la communication, le coût du transfert et le coût imprévisible du au fait que le processeur distant est en train d'exécuter un autre processus.

peut être classé dans la classe orienté-objet mais aussi dans les environnements se basant sur le concept de tâche. Nous allons donc essayer d'identifier un ou deux exemples pertinents de chaque classe, et de donner une explication des différents concepts centraux utilisés par ces environnements présentés. Avec comme objectif de rendre la description théorique des environnements plus tangible nous présentons quelques simples extrais de code, pris pour la plupart des manuels d'utilisation des environnements présentés.

2.3 Modèles basés sur le compilateur

Ce premier modèle reprend les approches de caractère implicite de programmation parallèle. Le programmeur n'a donc pas de soucis pour structurer son programme selon un paradigme parallèle. Par contre, c'est le compilateur qui analyse le programme et essaye de procéder à une parallélisation "automatique". Cependant, cette parallélisation automatique a montré certaines difficultés ; le code parallèle produit n'est pas très efficace, vu que l'approche ne peut pas utiliser les compilateurs standards optimisés selon l'architecture, comme un compilateur C classique l'est.
Cette approche est souvent utilisée si on veut paralléliser de petits programmes simples. Néanmoins, elle a montré qu'une structuration parallèle automatique est difficile à faire, due au manque d'informations que le compilateur a sur la nature de l'application, mais pour l'optimisation de synchronisation entre différents processus parallèles l'efficacité d'une approche automatique a été prouvée.

HPF (High Performance Fortran)

L'exemple classique de ce modèle est le *High Performance Fortran*(HPF) [23]. L'HPF est un standard informel[3] d'une extension de Fortran 90 et 95. La première version pour Fortran 90 a été développée pendant des conférences américaines entre des équipes de recherche et de constructeurs en 1992. Sa deuxième version date de 1996 et contient des adaptations pour Fortran 95.
L'HPF se base sur la parallélisation des données sur mémoire distribuée et utilise principalement le modèle SPMD. Donc chaque processeur exécute le même programme sur une partie différente des données[4].

Pour aider le compilateur à la parallélisation, le programmeur peut indiquer des directives pour la distribution des données. En effet, dans HPF un code séquentiel Fortran est complété par des directives `!HPF$` que le compilateur HPF peut mais ne doit pas nécessairement utiliser pour produire un programme parallèle. Spécialement, il est possible de définir une grille conceptuelle de processeurs :

```
!HPF$ PROCESSORS DIMENSION(2,2) :: P
```

signifie, que P est construit de 2x2 processeurs. Une deuxième définition concerne la distribution des données sur une grille conceptuelle :

```
      REAL DIMENSION(12) :: A
!HPF$ PROCESSORS DIMENSION(4) :: P
!HPF$ DISTRIBUTED (BLOCK) ONTO P :: A
```

signifie, que les données A sont distribuées d'une manière `BLOCK` (figure 2.1 a)), c'est à dire en continu et de taille identique sur chaque processeurs de P. Par cette distribution en blocs le processeur $p1$ de P reçoit les 3 premiers éléments de A le processeur $p2$ les

[3]HPF est un standard non officiellement spécifié.

[4]Le modèle SPMD utilisé pour le HPF diffère un peu de l'explication donnée au chapitre précédent. En effet, la copie des programmes et la communication entre ces copies se fait en HPF d'une manière implicite en se basant sur la parallélisation des données, lui, explicite.

trois suivants et ainsi de suite. Une autre manière de distribuer se fait par une approche cyclique (`CYCLIC`) (figure 2.1 b)). Dans l'exemple précédent les éléments 1, 4 et 7 de A étaient attribués au processeur $p1$. En plus, il est possible d'aligner différentes données pour minimiser la communication, en HPF toujours implicite, entre les processeurs :

```
!HPF$ ALIGN A(:) WITH B(:)
```

signifie que pour chaque i, $A(i)$ et $B(i)$ se trouve sur le même processeur.

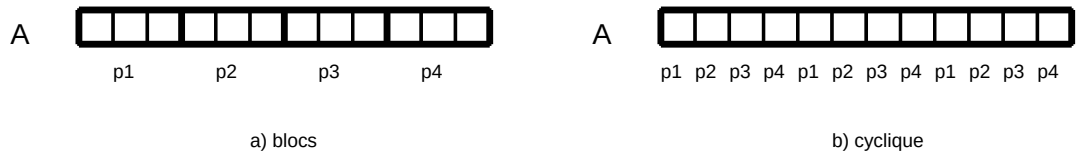


FIG. 2.1 – *Schéma d'une distribution en blocs a) et d'une distribution cyclique b) des données sur 4 processeurs (p1,p2,p3 et p4)*

La parallélisation se base alors sur des indications d'exécution. Par exemple, le `forall` va provoquer l'exécution en parallèle sur les différents processeurs suivants de la distribution des données :

```
FORALL (i=1 :n) A(i) = B(i)+1
```

Surtout pour des applications irrégulières le HPF montre de faibles performances. Ces problèmes d'HPF sont engendrés par un manque d'expressivité au niveau du parallélisme ainsi que la non adaptation à des classes de problèmes qui ne peuvent pas être résolus par un parallélisme des données. L'HPF n'est jamais devenu, malgré de grands efforts, un standard dans l'industrie. En effet, ce langage est en voie de disparition et son utilisation est souvent remplacée par OpenMP (voir en-dessous).

En résumé, un tableau avec nos exigences :

Exigences	**Généralité**	**High-level**	**Efficacité**	**Expressivité**	**Mod., Scal. et Réut.**
HPF	uniquement parallélisme des données	pas d'équilibrage de charge et très difficile pour des applications irrégulières	pas de compilateurs optimisés	pas de parallélisme vraiment explicite	code séquentiel Fortran avec directives

2.4 Modèles basés sur la coordination par message

Un modèle de coordination définit le mécanisme utilisé pour coordonner différents processus dans un programme parallèle. La coordination est ici explicite par message et intervient souvent à plusieurs étapes dans un programme. Le paradigme de base de ce modèle est donc le *message-passing*[5].
Les modèles de coordinations sont en général des extensions d'un langage séquentiel bien connu, et sont souvent implémentés sous forme de bibliothèque. Ainsi, ils sont utilisables sur la plupart des architectures MIMD.

[5]Voir aussi chapitre 1 à la page 17.

2.4.1 Modèles de coordination *low-level* basés sur la communication

La communication (synchronisation et échange de données) entre les processus concurrents est réalisée par l'échange de messages. La gestion de la coordination est donc à la charge du programmeur. Pour ce modèle *low-level*, c'est à dire toute la gestion des processus est à charge du programmeur, nous pouvons citer le *message passing interface* (MPI) et le *parallel virtuel machine* (PVM) comme environnements pertinents.

A) PVM (Parallel Virtual Machine)

Le développement de PVM [42] [38] a commencé en 1989 dans le laboratoire de recherche américain *Oak Ridge National Laboratory*. Le professeur Vaidy Sunderam et son équipe voulait faire des recherches sur le calcul distribué hétérogène. Pour cela ils avaient besoin d'un framework et ils ont inventé le concept d'une machine parallèle virtuelle. L'objectif d'un système PVM était de faire coopérer via un réseau LAN un ensemble hétérogène d'ordinateurs afin d'effectuer du calcul parallèle avec une puissance semblable au super-ordinateur.

Le but de PVM est donc de rassembler un ensemble d'ordinateurs via un réseau local afin de former un ordinateur virtuel parallèle. Les développeurs de PVM ont crée un daemon[6], qui doit être lancé sur chaque ordinateur qui participe, afin de créer ce ordinateur virtuel. Après le lancement de cet ordinateur virtuel les programmes PVM peuvent être exécutés par les utilisateurs.
Un programme PVM est un programme C ou Fortran avec quelques primitives d'une bibliothèque supplémentaire. Ces primitives rendent la gestion et la communication des différentes parties de cet ordinateur virtuel programmable. Pour utiliser les appels PVM il suffit donc d'inclure le fichier bibliothèque `pvm.h` au début du code C ou Fortran.

Pour écrire un programme parallèle, des processus doivent être exécutés par différents processeurs. En PVM nous atteignons ce but par un premier appel explicite de :

```
pvm_spawn(‘‘mon_prog’’, NULL, PvmTaskDefault, 0, n_proc, pids)
```

cela lance `n_proc` copies du programme `‘‘mon_prog’’` sur des processeurs qui forment l'ordinateur virtuel. Un nombre entier identificateur de chaque processus lancé est retourné dans le tableau `pids`. Cet appel retourne aussi le nombre de processus effectivement lancés qui peut être différent de `n_proc`.
Afin, de pouvoir implémenter le modèle SPMD l'environnement PVM fourni aussi un appel :

```
pvm_mytid()
```

qui retourne l'entier identificateur du processus qui exécute une copie programme. Ou encore l'appel :

```
pvm_parent()
```

qui retourne une valeur pour identifier le processus père ou racine, c'est à dire celui qui lance l'appel pvm_spawn initial.
Remarquons encore, qu'en PVM il est possible de changer le nombre d'ordinateurs participant à l'ordinateur virtuel. Des ordinateurs peuvent être ajoutés ou retirés pendant l'exécution de PVM. Ce qui rend possible une gestion limitée d'un équilibrage de charge.

Comme déjà dit au chapitre précédent dans un environnement *message-passing* la coordination effectuée par message est explicite par une primitive d'envoi et une de réception.

[6] Un deamon PVM est un processus "invisible" qui prend en charge l'exécution des programme PVM sur le processus où il est lancé.

PVM met à disposition plusieurs primitives pour l'envoi :

```
pvm_initsend(PvmDataDefault) ;
pvm_pack(buf) ;
pvm_send(pid, msgtag) ;
```

Si on veut envoyer un message d'un processus A à B, le processus A doit d'abord appeler `pvm_initsend()`. Cela initialise le buffer d'envoi par défaut et spécifie l'encodage du message. Après cette initialisation, A doit emballer `pvm_pack()` toutes les données à envoyer dans le buffer d'envoi. Enfin, A peut envoyer le message avec `pvm_send()` en spécifiant l'identificateur `pid` du destinataire B et un `msgtag` qui permet à B d'identifier quelle sorte de message il reçoit.
Les primitives pour la réception sont similaires :

```
pvm_recv(pid, msgtag) ;
pvm_unpack(buf) ;
```

Ici, B appelle `pvm_recv()` en spécifiant l'identificateur de l'expéditeur A ainsi que le `msgtag` du message attendu. Puis B procède à un déballage des données via `pvm_unpack()`. Remarquons encore, que les appels d'envoi et de réception sont par défaut bloquants en PVM.

Une autre possiblité en PVM c'est de pouvoir nommer des ensembles de processus et d'effectuer des opérations de broadcast `pvm_bcst()` ou de synchronisation `pvm_barrier()` à l'intérieur d'un tel ensemble.

Voici, un programme "Hello World" en PVM :

```
/*Programme helloPVM*/

if (pvm_parent() == PvmNoParent) {     /*si processus racine*/

   /*demande du nombre de processus*/
   printf("Input Sub-Proc Total: ");
   scanf("%d", &sub_proc_total);
   /*lancement de nombre de processus*/
   num = pvm_spawn("helloPVM", (char**)0, 0,
                  "",sub_proc_total,tids);

   for(i=0; i<num; i++)     /*pour chaque processus lancé*/
   {
      /*réception d'un message*/
      buf_id = pvm_recv(-1, tag);
      pvm_upkstr(buf);    /*"unpack" pour les strings*/

      printf("%s\n", buf);
   }

} else {     /*si processus lancé*/
   /*envoi d'un message*/
   sprintf(buf, "Greetings from task:%d",pvm_mytid());
   pvm_initsend(PvmDataDefault);
   pvm_pkstr(buf);     /*"pack" pour les stings*/
   pvm_send(ptid, tag);
}

pvm_exit();   /*fermeture de l'environnement PVM*/
```

B) MPI (Message Passing Interface)

Par rapport à PVM, qui était développé à l'intérieur d'une équipe de recherche, MPI [35] [41] a été spécifié par un group de 40 experts académiques et industriels pendant une série de conférences en 1993-1994. En 1997 le premier standard a vu une extension en spécifiant une deuxième version MPI-2. L'objectif des conférences initiales était de définir une API standard pour le *message-passing* afin de pouvoir développer des programmes portables sur différentes types d'architecture. Aussi, les constructeurs de machines s'engageaient de développer des implémentations efficaces du standard sur leurs machines. A cause de cela, les implémentations de MPI sont les environnements les plus efficaces et les plus répandus en domaine du calcul parallèle. Par exemple, les distributions comme MPICH [36], LAM [31] ou WMPI [50] (MPI pour Windows) fournissent des implémentations pour presque chaque architecture.

Comme PVM, les distributions de MPI implémentent le standard dans des bibliothèques que le programmeur peut inclure dans le code de ses programmes Fortran, C ou C++. Aussi, il existe des projets qui essayent d'implémenter MPI pour Java, mais ces projets sont souvent limités par la faible performance sous-jascente du langage Java.
Le standard MPI définit un ensemble assez complet de plus de cent-cinquante fonctions. Mais, en général une petite dizaine suffit pour la conception normale d'un programme parallèle.

Au lieu de lancer des deamons comme en PVM, MPI fournit directement des commandes pour son environnement :

```
MPI_Init(argc, argv) ;
```

pour l'initialisation de l'environnement MPI ;

```
MPI_Comm_size(contexte_communication, size) ;
```

renvoie dans `size` le nombre de processus MPI participant au contexte de communication ;

```
MPI_Comm_rank(contexte_communication, my_rank) ;
```

renvois dans my_rank le rang du processus, c'est à dire un entier identificateur du processus ;

```
MPI_Finalize() ;
```

termine un environnement MPI. Le context_communication spécifie un ensemble de processus MPI. Par exemple, le contexte MPI_COMM_WORLD comprend par défaut tous les processus connus par l'environnement MPI.
Remarquons qu'en général en MPI nous avons un processus MPI par processeur. Dans la première version initiale de MPI il n'existait pas de possibilité pour créer des processus MPI dynamiquement, c'est à dire on ne pouvait pas ajouter un processus pendant l'exécution d'un programme MPI. Donc l'environnement MPI, à l'inverse de PVM, est statique au niveau du nombre de processus pendant l'exécution d'un programme MPI[7]. Ainsi, un programme MPI ne peut non plus s'adapter pendant son exécution à un changement de l'architecture physique sous-jascente.

Pour la communication point à point MPI fournit une large gamme de primitives. Les deux les plus importantes sont les primitives d'envoi et de réception (par défaut bloquant et asynchrone) :

```
MPI_SEND(buf, taille, MPI_type, rank_dest, msgtag, contexte_communication) ;
```

[7] En MPI-2 le standard a été complété avec des fonctionnalités de création dynamique (*spawn*) comme PVM les possède.

```
MPI_RECV(buf, taille, MPI_type, rank_src, msgtag, contexte_communication,
        stat) ;
```

En MPI on a regroupé les différentes étapes d'un échange de message de PVM en une primitive pour l'envoi et une pour la réception. Ces primitives permettent l'échange de messages entre deux processus. Dans le cas où A veut envoyer un message à B, A fait un appel à `MPI_send` et met dans `buf` le contenu du message. De son tour B doit faire un appel explicite à `MPI_Recv`. Au retour de ce appel le `buf` de B contient le contenu du message envoyé par A.
Pour le type de données transmises dans le contenu du message, donc contenu dans `buf`, MPI spécifie des types standards `MPI_type`. Par exemple pour les entiers le type MPI s'appelle `MPI_INT`. Si l'utilisateur doit envoyer des structures spéciales MPI définit des fonctions qui permettent de créer de nouveaux types MPI utilisateurs.
Les paramètres `rank` contiennent respectivement l'identificateur destination pour le `MPI_send` et l'identificateur expéditeur pour le `MPI_recv`. Ainsi `context_communication` définit l'ensemble des processus MPIqui peuvent s'échanger des messages. En effet deux processus doivent appartenir au même contexte pour pouvoir communiquer. `msgtag` est utilisé comme en PVM pour différencier différente type de message. `stat` contient des informations supplémentaires sur le message.
Remarquons que MPI spécifie à coté de ces primitives bloquantes et asynchrones entre autre des primitives de communication non-bloquantes ou même synchrones.

En plus de ces communications point à point, MPI propose aussi des primitives de communication collectives. Comme déjà énoncé, pour que deux processus MPI puissent communiquer ils doivent appartenir à un même contexte de communication. Remarquons qu'un processus peut appartenir à plusieurs contextes de communication. Aussi, pour les communications collectives de MPI les contextes de communication définissent l'ensemble de processus MPI qui participent à cette communication. Par exemple, dans une diffusion générale (*broadcast*) tous les processus d'un contexte de communication sont alors informés. Néanmoins, la définition de ces contextes de communication d'MPI ne se limite pas uniquement à définir un ensemble de processus pour la communication. Ainsi, MPI propose des appels qui permettent de définir des contextes de communication selon une conception logique de la topologie des processus qui colle bien à l'architecture réelle des processeurs. Cette conception selon une topologie qui respecte l'architecture réelle a pour objectif de faire des optimisations pour la communication entre autre des opérations collectives. Par exemple, MPI permet de définir des grilles cartésiennes (*mesh*) qui permettent de minimiser le temps et le nombre de communications réelles à effectuer pour une communication collective. Pour notre diffusion générale d'une information chaque processus envoie alors l'information reçue à ses voisins et ainsi de suite et on peut construire un sorte de arbre de diffusion optimale à partir de la source.
Il existe trois primitives importantes pour la communication collective :

```
MPI_Barrier(contexte_communication) ;
MPI_Bcast(buf, taille, MPI_type, rank_src, contexte_communication) ;
MPI_Reduce(buf_operand, buf_result, taille, MPI_type, operation,
           rank_dest, contexte_communication) ;
```

Notons d'abord pour ces trois primitives qu'une primitive collective en MPI est toujours bloquante. C'est à dire qu'un appel à une primitive collective bloque jusqu'au moment où tous les processus qui participent à la communication ont fait appel à la même primitive. Dans une communication collective tous les processus du contexte de communication sont donc bloqués.
La première primitive `MPI_Barrier` a le simple but de synchroniser tous les processus du contexte de communication. Il utilise donc l'effet bloquant des primitives collectives

La diffusion générale d'une information (*broadcast*) est réalisée par a primitive `MPI_Bcast`. Pour la sémantique de cette primitive, le contenu du buffer `buf` du processus `rank_src` est diffusé vers tous les processus du contexte de communication `contexte_communication`. Quand l'opération *broadcast* clôture le buffer `buf` des processus destinataires contiennent l'information diffusée. Les paramètres `taille` et `MPI_type` ont la même signification que dans les primitives point à point. Remarquons, qu'il existe des primitives de diffusion générale en MPI qui permettent d'envoyer à chaque processus seulement une partie de l'information initiale. Ainsi, nous pouvons implémenter le partage de données en MPI.
Le dernier appel présenté ici est `MPI_Reduce`. Il permet de faire un calcul groupé (*reduce*) sur des opérants que les processus du contexte de communication fournissent. Ainsi, les processus fournissent ces opérants dans le buffer `buf_res` et l'appel à cette primitive va alors effectuer sur ces opérants l'opération spécifiée dans le paramètre `opration`. Par exemple, MPI_SUM, une opération collective définit par le standard MPI, fait la somme de tous les opérants est . Finalement, l'appel de la primitive *reduce* se termine en plaçant le résultat du calcul dans `buf_res` de l'unique processus destinataire `rank_dest`.

Voici, un programme "Hello World" en MPI :

```
/*Programme helloMPI*/

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if(my_rank == 0){   /*si processus racine*/
   for (source=1; source<size; source++) {
       /*réception d'un message*/
       MPI_Recv(message, 256, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
       printf("%s\n", message);
   }

}else{   /*si processus non-racine*/
   /*envoi d'un message*/
   sprintf(message, "Greetings from process %d!", my_rank);
   MPI_Send(message, strlen(message)+1, MPI_CHAR, 0,
            tag, MPI_COMM_WORLD);
}

MPI_Finalize();
```

C) Quelques différences entre MPI et PVM

Malgré la forte ressemblance entre PVM et MPI nous identifions plusieurs différences [10] [38] significatives.
La machine virtuelle parallèle PVM implémente un contrôle de processus, qui permet de lancer et d'arrêter des processus ou de vérifier s'il sont en exécution. En plus, PVM permet d'ajouter ou de supprimer des processeurs ou ordinateurs pendant son exécution. Ainsi, il dispose un schéma basique de notification qui informe les processus, via des messages spéciaux, d'un changement de l'architecture réelle. Ce comportement dynamique inhérent de PVM permet une gestion des ressources et de la tolérance aux pannes. Toutefois cette gestion dynamique de PVM est limitée. La première version du standard MPI ne disait rien sur la création dynamique des processus, la deuxième version MPI-2 essaye de combler cette défaillance. Néanmoins, MPI est de nature statique et aucune gestion dynamique des ressources n'est possible.

En MPI un ensemble de processus peut être organisé dans une interconnexion topologique logique (voir 31). Cette topologie devrait correspondre à l'architecture réelle sous-jascente, afin d'augmenter la performance des communications. PVM ne dispose pas d'un tel niveau d'abstraction.
Finalement, la notion des contextes de communication pour un ensemble de processus en MPI, permet la création de bibliothèques dans un style *message-passing* qui peuvent protéger ou cacher leurs messages internes afin qu'ils ne soient pas reçus par les applications utilisateurs.

Comparons PVM et MPI au niveau des exigences :

Exigences	**Généralité**	**High-level**	**Efficacité**	**Expressivité**	**Mod., Scal. et Réut.**
PVM	applicable sur les machines MIMD	gestion dynamique limitée des ressources	utilisation des compilateurs standards optimisés	parallélisme explicite avec lancement de processus (spawn)	programmes séquentiels modifiés (appels système et gestion de la machine virtuelle parallèle)
MPI	standard applicable sur toutes les machines MIMD	pas d'équilibrage de charge et aucune gestion dynamique possible	support des constructeurs et possibilité d'optimisation des communications via définition de topologie	parallélisme explicite	programmes séquentiels modifiés (appels système)

2.4.2 Modèles de coordination *high-level* basés sur la communication

Dans ce modèle la communication entre les processus est aussi réalisée par l'échange de messages, mais un gestionnaire de l'environnement prend en charge différents aspects de la coordination "*high-level*". Par exemple, ce sont des environnements qui libèrent le programmeur de la gestion des affectations des processus aux processeurs, des optimisations pour la communication comme des appels non-bloquants, des aspects d'équilibrage de charge et gestion de ressources, de la tolérance aux pannes, ...

A) Charm

L'environnement Charm [13] a été développé par un groupe de recherche du *Parallel Programming Laboratory* de l'université de *Illinois at Urbana-Champaign* fin des années quatre-vingts. Charm est une bibliothèque d'extension du langage C et fournit surtout des méthodes d'invocation asynchrone à distance. Ces dernières sont en effet à la base du concept *message-driven* dont Charm était une des premières implémentations. Pour cela ce langage dispose aussi d'une sorte d'objet appelé *chare* par lequel il est possible de définir les méthodes d'entrée (*entry methods*). Ces méthodes d'entrée peuvent être appelées via des messages par des processus distants et ont la caractéristique d'être asynchrones, c'est à dire qu'ils ne fournissent pas de résultat en retour. Pour retourner un résultat il suffit que la méthode d'entrée appelée, appelle à son tour une méthode d'entrée du *chare* du processus distant avec un message contenant le résultat (voir aussi 22).
L'environnement Charm n'est pas vraiment un environnement *high-level* car il ne dispose pas de facilités de programmation, mis à part des optimisations de communication dues aux appels asynchrones (plus d'attente d'un retour) et de la suppression de l'appel ex-

plicite d'une primitive de réception chez le destinataire (plus d'attente d'une arrivée de message) dans le *message-driven*. Mais, nous le citons dans cette section, car l'environnement Charm constitue la base de CHARM++ (voir en-dessous), qui dispose des facilités de programmation comme l'équilibrage de charge dynamique.

B) AMPI (Adaptative MPI)

L'AMPI [11] [1] est une implémentation du standard MPI au-dessus de l'environnement CHARM++[8] qui est décrit en-dessous. Cet environnement a été développé à partir de 2001 au même laboratoire que Charm et CHARM++ à l'université de *Illinois at Urbana-Champaign*. Particulièrement, AMPI utilise la capacité d'équilibrage dynamique de charge de CHARM++. Cette capacité se base sur le support d'exécution CONVERSE qui est l'environnement sous-jascent commun de AMPI et CHARM++.
En résumé, AMPI fournit donc une implémentation de MPI avec des capacités d'équilibrage dynamique de charge. Il n'est donc pas nécessaire de réécrire un programme parallèle dans un langage adapté aux équilibrages dynamiques mais il suffit de reprendre le programme MPI et de le compiler en incluant la bibliothèque AMPI.

Pour utiliser les avantages du *message-driven* de CHARM++ l'environnement transforme les processus MPI dans des *user-level thread*, qui ne sont rien d'autre qu'une spécialisation des objets *chare* de Charm. Ainsi, il est possible de placer plusieurs "*threads*" sur un seul processeur. Ce principe s'appelle virtualisation [28] des processeurs (figure 2.2). En effet, dans des programmes MPI traditionnels, un processus MPI est affecté à un processeur réel. Un *user-level thread* d'AMPI peut alors être vu comme un processus affecté sur un processeur virtuel. Cette virtualisation des processeurs réels possède différents avantages :

- **Surcharge des temps de communication par des calculs :** Si un processeur virtuel est bloqué par une primitive de réception, un autre processeur virtuel sur le même processeur réel peut être exécuté. Ainsi, on augmente l'occupation du processus réel et on élimine le besoin des programmeurs de spécifier de la surcharge statique de communication par du calcul, comme c'est le cas dans un programme MPI traditionnel.
- **Équilibrage de charge dynamique :** Si un processeur réel devient surchargé, le support d'exécution CONVERSE d'AMPI peut migrer quelques processeurs virtuels vers un processeur réel moins chargés. CONVERSE prend ces décision d'équilibrage sur des mesures automatiques des charges. Nous traiterons le système d'équilibrage de charge de CONVERSE dans la section suivant, qui traitera CHARM++.
- **Appels non-bloquant :** Comme nous l'avons vu les appels de MPI sont bloquants. Avec le principe des processeurs virtuels, un processeur réel passe à un autre processeur virtuel pendant que le premier processus MPI est bloqué par un appel. Ainsi, l'attente de communication bloquant est combler par des calculs d'autres processeurs virtuels et les appels bloquant semble d'être des appels non-bloquants.
- **Utilisation flexible des ressources :** La capacité de pouvoir migrer des processeurs virtuels peut être utilisée pour adapter la charge si la réalité physique de l'architecture réelle change.

Comme AMPI est une implémentation du standard MPI tous les appels MPI peuvent être conservés. Néanmoins, la conversation d'un programme MPI en AMPI, par exemple pour profiter de l'équilibrage de charge, nécessite quelques petites modifications.

[8] En réalité, AMPI est construit avec la capacité de créer des *user-level thread* de CHARM++ et non pas sur le langage CHARM++.

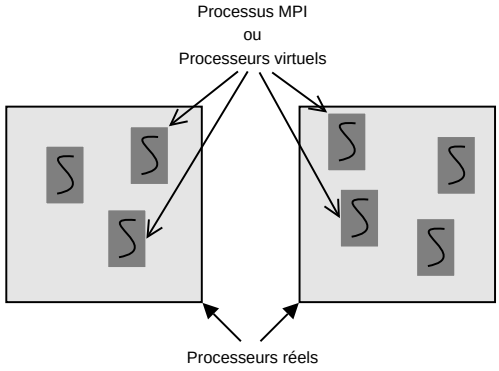


FIG. 2.2 – *Principe de virtualisation : Un processus* MPI *est implémenté comme un user-level thread, plusieurs user-thread son affecté sur un seul processeur réel.*

Il existe en AMPI plusieurs *user-level threads* par processeur réel. Il peut donc avoir interférence entre ces différents *threads*. Plus spécifiquement, des processus MPI peuvent utiliser des variable globales. Une variable globale est une variable qui est stockée à une place fixe en mémoire et peut être accédée par tous les processus du processeur. Comme dans les programmes MPI traditionnels on affecte un processus à un processeur, les processus MPI suppose qu'ils peuvent utiliser ces variables globales indépendamment. Ainsi, une variable globale peut avoir deux valeurs différentes sur deux processeurs. Une majeure transformation pour un code AMPI est donc de remplacer les variables globales par des variable locales, pour les protéger d'accès concurrents. Pour cela, il suffit de placer toutes les variables globales dans une structure, puis, de créer une variable locale avec ce type de structure, de passer cette variables locales au sous-routines et de remplacer les appels des différentes variables globales par un appel d'une variable de cette structure de données.

Comme déjà dit, l'équilibrage de charge et spécialement la migration demande une attention particulière dans la transformation en AMPI. En effet, pour migrer un *thread* d'un processeurs à un autre il est important de migrer aussi son état actuel. Ce état pour un *user-level thread* d'AMPI est caractérisé par ces variables locales (*stack data*), quelques registres du processeur et les variables allouées dynamiquement (*heap data*). Remarquons, que la transformation de variables globales en variables locales facilite déjà la migration.
Pour les registres il suffit de les copier vers le nouveau processeur. Malheureusement, le *stack* (pile) est plus difficile à migrer. Dans une machine à mémoire distribuée si on déplace le *stack* vers la mémoire du nouveau processeur, il est probable qu'il suit alloué à une nouvelle location. Ainsi, des pointeurs vers l'ancien *stack*, par exemple les *frames* pointeur, ne sont plus valides. Il est impossible d'actualiser tous ces pointeurs du *stack* car il se trouve dans des registres et à des régions du *stack* dont la structuration dépend fortement de la machine et du compilateur utilisée. La solution est de migrer le *stack* de telle manière qu'il occupe la même location qu'avant la migration. Heureusement, tous les systèmes d'exploitation actuels supportent la mémoire virtuelle. Pour assurer que chaque *stack* peut être alloué à la même place dans toutes les mémoires, on divise la mémoire virtuelle disponible en p régions et on affecte une région par *thread*. Cette approche s'appelle *iso-adresse* et est aussi utilisé dans l'environnement PM2.
Pour les variables allouées dynamiquement (*heap*), seul l'application utilisateur a la connaissance de leur localisation. Dans les dernières versions d'AMPI la même stratégie *iso-adresse* est aussi utilisée pour les allocations des données allouées dynamiquement. Ici

AMPI remplace la fonction d'allocation de mémoire *malloc*[9] par une propre version qui alloue la mémoire d'une tâche dans une zone réservée à la tâche. Ainsi, l'environnement connaît l'emplacement des données du *heap* et il suffit pour la migration de copier cette zone réservée vers le nouveau processeur. Vu la taille que des données du *heap* peuvent prendre, cette méthode limite néanmoins fortement la mémoire disponible [10], car pour chaque *thread* on doit réserver une région par processeur. Une alternative est de enregistrer chez le support d'exécution d'AMPI les données allouées dynamiquement. Ainsi, la bibliothèque AMPI fourni un appel :

```
MPI_Register(variable, (MPI_PupFn) fonctPup);
```

qui permet d'enregistrer les variables allouées dynamiquement qui doivent être migrées avec le *thread*. Avec cet enregistrement le support d'exécution peut localiser les données du *heap* mais il ne sait pas quelles tailles et structure ces données ont. Ainsi, on doit encore fournir une fonction *pup* qui permet au support de traiter les variables allouées dynamiquement. Cette fonction est appelée avant et après la migration. Avant la migration la fonction *pup* sérialise la variable, c'est à dire la variable est transformée pour pouvoir la transporter sur le réseau, et *pup* se charge de libérer la mémoire allouée. Après la migration ,*pup* réagit de façon inverse. Elle alloue la mémoire nécessaire chez le nouveau processeur et rétablit la structure initiale de la variable (*unpack*). Un exemple concret d'une telle fonction *pup* se trouve au chapitre 4.
Néanmoins, pour la migration, CONVERSE pose certaines limites. Les fichiers, sockets et signaux d'un *thread* ne peuvent pas être migrés avec. Mais, les *user-level threads* d'AMPI sont uniquement migrés si le programme utilisateur a fait un appel à une autres primitives de la bibliothèque AMPI :

```
MPI_Migrate();
```

Cette primitive doit être appelée, comme une communication collective, sur tous les processus AMPI. Elle incite le support d'exécution CONVERSE de vérifier des données statistiques mesurées sur l'exécution des *threads* et/ou la charge des processeur afin de faire un équilibrage de charge dynamique et des migrations en cas de besoin. Donc l'appel à MPI_Migrate() ne provoque pas nécessairement des migrations, mais incite plutôt le support d'exécution à tester si des migrations seraient utiles. La décision de migration est uniquement faite par des stratégies de migration. Des informations supplémentaires de CONVERSE sur l'évaluation des données récoltées et les stratégies de migration proposées se trouvent dans la section suivante sur CHARM++. Comme un *thread* peut être uniquement migré si le programme utilisateur a fait appel à la primitive de migration, le programmeur peut s'assurer qu'à ce moment précis il n'y a pas des fonctions non-migrables utilisées par le *thread*. Les techniques de migration de CONVERSE posent encore des problèmes entre des architectures hétérogènes. Ces problèmes sont dus entre autre au *iso-adresse* de CONVERSE qui ne tient pas compte des structurations différentes du *stack* ou des registres machines sur différentes architectures.

À parts quelques primitives supplémentaires pour la migration, un code AMPI est donc fort semblable à un code MPI classique. Pour un exemple pratique, d'AMPI, nous renvoyons le lecteur aux chapitres 4 et 5 qui traiteront de l'AMPI ainsi que la transformation d'un programme MPI en AMPI d'un point de vue pratique.

[9] En AMPI on parle alors aussi d'*isomalloc*.

[10] Sur les nouvelles machines à 64-bit cette limitation est moins critiques car la mémoire virtuelle est beaucoup plus grands.

En résumé, un tableau avec nos exigences d'AMPI :

Exigences	Généralité	High-level	Efficacité	Expressivité	Mod., Scal. et Réut.
AMPI	applicable sur les machines MIMD	équilibrage dynamique de charge	virtualisation des processeurs	parallélisme explicite	code MPI avec adaptation pour la migration

2.5 Modèles basés sur le concept d'objet

Dans ce modèle, la théorie des objets est utilisée pour fournir l'abstraction de la concurrence. Les environnements de ce modèle fournissent des cadres, en général *high-level*, dans lequel des programmes parallèles peuvent être développés. Comme exemple de ce modèle citons ici uniquement Charm++.

Charm++/Converse

En 1993, l'équipe de recherche du *Parallel Programming Laboratory* de l'université *Illinois at Urbana-Champaign* a transformé l'environnement Charm (voir au-dessus) en Charm++. Charm++ est complètement orienté-objet et reprend le concept de *message-driven* de Charm. Charm++ est un langage qui se base principalement sur la syntaxe de C++. Au cours du temps, la structure de Charm++ a été modifié. Notamment, les chercheurs ont décidé de diviser l'implémentation de Charm++ entre le langage Charm++ proprement dit et son support exécution. L'indépendance de ce support exécution, appelé maintenant Converse, rendait, entre autre possible des implémentations du standard MPI et PVM au-dessus de se support indépendamment du langage Charm++. Ainsi, l'environnement AMPI[11] est implémenté au-dessus de Converse.
En 1999, Converse a été re-implémenté totalement en C++. Depuis 2000, ce support d'exécution dispose un système d'équilibrage dynamique de charge qui permet entre autre de faire des migrations. Récemment, les chercheurs de Charm++ ont implémentés des fonctionnalités de *checkpointing* à Charm++. Avec le *checkpointing* il est possible d'enregistrer l'état actuel d'un processus afin de pouvoir relancer le processus à partir de cet état à un moment antérieur pour une gestion de tolérance aux pannes ou de migration.

a) Le langage Charm++

En premier lieu, le langage Charm++ [1] [14] est une librairie orientée objet de programmation parallèle pour C++ portable sur la plupart des machines parallèles. Charm++ fait partie des langages orienté-objets augmentés de quelques appels système et utilise le concept *message-driven*. Ainsi, des calculs sont lancés lors d'arrivées de messages. À leur tours, les calculs peuvent envoyer des messages, si possible vers d'autres processeurs, ce qui provoque d'autres calculs.

Un programme Charm++ est composé de méthodes séquentielles écrites en C++ auxquelles sont ajoutées des structures qui permettent l'utilisation de ces sous-routines dans un environnement parallèle. Un objet Charm++, appelé *chare*, est alors une sorte d'objet C++ avec des méthodes d'entrée (entry methods) qui peuvent être invoquées à partir d'un *chare* distant. Charm++ utilise une méthode d'invocation asynchrone à distance. Comme les méthodes d'entrée d'un *chare* distant ne peuvent pas être appelées via un pointeur ordinaire, Charm++ utilise l'intermédiaire d'un *proxy*. C'est ce *proxy* qui va s'occuper de relayer le message à la méthode d'entrée appelée du *chare* distant. Pour que

[11] Remarquez que AMPI utilise néanmoins quelques fonctionnalités de Charm++.

le support d'exécution de CHARM++ puisse transmettre les messages, il est nécessaire de définir une interface notamment avec les méthodes d'entrée. Le CHARM++ *interface translator* permet alors d'enregistrer ces méthodes d'entrée dans le support d'exécution et spécifie en même temps une déclaration pour le ou les proxy's. Voici un exemple d'un tel fichier interface :

```
// File: pgm.ci

mainmodule Hello {
   readonly CProxy_HelloMain mainProxy;
   mainchare HelloMain {
      entry HelloMain();
      entry void PrintDone(void);
   };
   group HelloGroup {
      entry HelloGroup(void);
   };
};
```

Chaque programme CHARM++ commence avec la création d'un *mainchare*[12] sur un processeur 0 en appelant les constructeurs de ces *chares*. Typiquement, ce *chare* initial va créer d'autres *chares* sur d'autres processeurs qui vont s'exécuter simultanément pour résoudre le problème du programme. Pour la création d'un *chare* le programmeur n'a pas besoin de spécifier un processeur. C'est le support d'exécution CONVERSE qui va choisir le processeur le plus approprié afin d'équilibrer la charge (*seed-balancer*). Remarquons ici, qu'il est possible de créer un *chare* à distance, donc on doit déclarer le constructeur d'un *chare* comme une méthode d'entrée.
Semblable aux *user-level threads* d'AMPI les *chares* ne sont pas affectées à un processus spécifique et il est alors facile de les migrer. En plus, un objet comme un *chare*, contient toutes les données dont il à besoin pour le calcul, ce qui facilite la migration d'un objet *chare* par rapport aux *threads*. Le programmeur doit uniquement créer une méthode "*pup*", qui indique au support d'exécution comment sérialiser les données de l'objet avant la migration et comment les dé-sérialiser dans la mémoire du nouveau processeur après la migration de l'objet.

Comme déjà dit plusieurs fois, l'importance d'une invocation asynchrone résulte dans le fait de vouloir minimiser les temps d'attente dus aux communications. Ainsi, CHARM++ propose aussi des structures d'ensemble de *chares* plus évoluées pour augmenter la performance du système. Par exemple, il est possible de créer des tableaux de *chare* (*chare arrays*). Un tel tableau regroupe un ensemble de *chare* qui constituent alors ces éléments. Le tableau dispose un identificateur globale dans tous le système et chacun de ces éléments (*chares*) est accessible via son index. Avec l'identificateur globale on peut entre autre faire des diffusions générales (*broadcast*) ou des opérations collectives (*reduce*) sur les éléments du tableau.
L'affectation des éléments (*chares*) d'un tableau sur un processeur est effectuée par le support d'exécution. Les *chares* d'un tableau sont donc distribués sur l'architecture et c'est le support d'exécution qui fournit la transparence de localisation pour les éléments du tableau (voir figure 2.3). Spécialement, les tableaux sont utilisés pour l'équilibrage de charge. En effet, le support d'exécution peut créer et supprimer dynamiquement des chares d'un tableau sur des processeurs tout en garantissant que les messages pour ces *chares* arrivent toujours proprement (voir figure 2.3 (après migration)). Ainsi, un élément d'un

[12]Il peut en avoir plusieurs *mainchares* par programme CHARM++.

tableau peut être migré à tout moment.
Les groupes sont d'autres structures de *chares* qui assurent que des copies d'un *char* se trouvent sur chaque processeurs ou sur chaque noeuds de processeurs (*nodegroups*) et qui facilitent des implémentations basées sur une parallélisation de données.

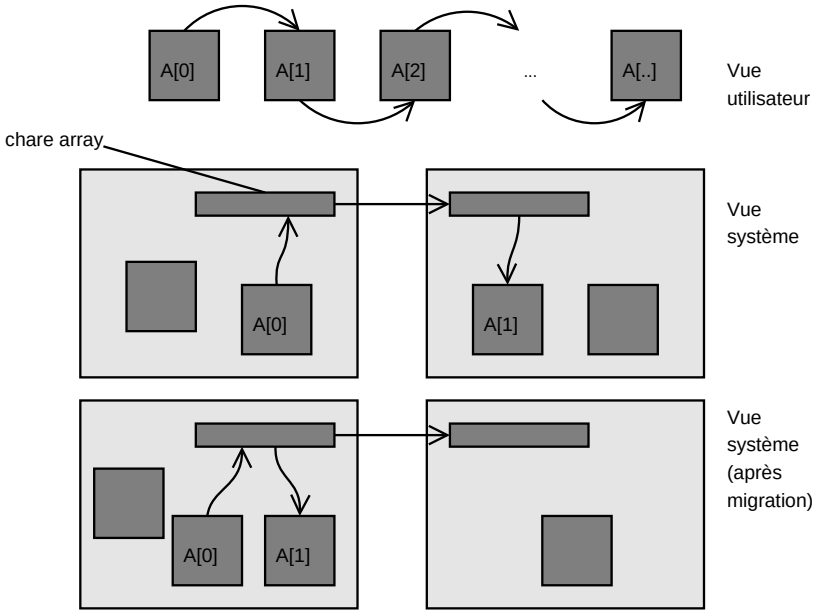


FIG. 2.3 – *Un tableau de chares, d'un point de vue utilisateur et d'un point vue système sur deux processeurs avant et après migration.*

Notons encore, que CHARM++ permet aussi d'exécuter les méthodes d'entrée dans des *user-level threads* séparés, afin d'étendre les fonctionnalités du support d'exécution. Par exemple pour implémenter un appel *synchrone*, une méthode d'entrée dans un *thread* peut alors bloquer en attendant des données d'une méthode d'un char distant qui retourne un résultat dans un message. C'est cette fonctionnalité de CHARM++ que AMPI utilise pour simuler les processus MPI classique.
Nous voyons déjà ici, que l'environnement CHARM++ commence à se compliquer. En fait, les développeurs ont ajouté avec le temps beaucoup de fonctionnalités à CHARM++. Toutes ces fonctionnalités le rendent assez complet et flexible pour une large gamme de problèmes. Mais cela rend aussi le langage assez complexe pour un débutant. Par exemple, nous avons parlé des invocations asynchrones à distance mais nous avons aussi dit que les méthodes d'entrée sont appelées via des messages. Les deux descriptions sont correctes. En effet, on trouve des appels distants avec la même syntaxe qu'un appel à une méthode locale[13]. Mais, on a aussi des appels à distance, qui ressemble plus à un envoi de message. Dans ce cas, CHARM++ propose des objets message qui englobent en fait une structure de données qui correspond aux paramètres d'une méthode entrée. Ainsi, il est par exemple, possible d'envoyer un Proxy d'un *chare* à un autre ou encore le support d'exécution peut englober, avec l'aide la fonction "*pup*", le *chare* lui-même dans un message afin de le migrer. Remarquons, qu'il existe en CHARM++ quatre sortes d'objet message pour différents propos.

[13]Sauf qu'on appel une méthode d'un Proxy.

Voici un exemple "hello world" d'un programme CHARM++ :

```
// File: helloCharm.h

#include "Hello.decl.h" // Créer avec le fichier pgm.ci
// "decl.h" déclare le Proxy

class HelloMain: public Chare {
 public:
   HelloMain(CkArgMsg *);
   void PrintDone(void);
 private:
   int count;
};

class HelloGroup: public Group {
 public:
   HelloGroup(void);
};

// File: helloCharm.C

#include "helloCharm.h"

CProxy_HelloMain mainProxy;

HelloMain::HelloMain(CkArgMsg *msg) {
   delete msg;
   count = 0;
   mainProxy=thishandle;
   CProxy_HelloGroup::ckNew(); // Créer un nouveau "HelloGroup"
}

void HelloMain::PrintDone(void) {
   count++;
   if (count == CkNumPes()) { // Attendre que tous les membres
                              // du groupe ont fait printf
     CkExit();
   }
}

HelloGroup::HelloGroup(void) {
   ckout << "Hello World from processor " << CkMyPe() << endl;
   mainProxy.PrintDone();
}

#include "Hello.def.h" // Créer avec le fichier pgm.ci
// "def.h" enregistre le proxy et les méthodes d'entrée chez Converse
```

***b) Le support d'exécution* CONVERSE et l'équilibrage de charge**

Le support d'exécution CONVERSE sur lequel CHARM++ mais aussi AMPI est construit a l'intention de contrôler la complexité de la programmation parallèle.
Comme déjà dit CONVERSE rend possible les appels à distance. Spécialement, c'est lui qui ordonnance le prochain message à traiter (*message sheduler*). Pour comprendre ce fonctionnement d'ordonnancement, nous pouvons imaginer un pool qui contient tous les messages non encore traités. Ignorons comment un tel pool pourrait être implémenté. Supposons aussi dans le cadre de CHARM++ qu'un *chare* s'est terminé et que son processeur

devient libre. Remarquons ici que l'exécution d'une méthode d'un *chare* est toujours non-préemptive, c'est à dire qu'il est impossible d'interrompre l'exécution de cette méthode. Pour notre processus libre, le support d'exécution CONVERSE choisit alors un message du pool des messages qui correspond à la méthode d'entrée d'un *chare* qui est hébergé sur ce processeur. Ce *chare* est alors activé sur le processeur et sa méthode d'entrée en question est exécutée. Ainsi, CONVERSE peut diminuer le temps d'inoccupation d'un processeur, pour autant qu'il y ait un *chare* avec un message correspondant disponible. C'est a ce niveau d'ordonnancement des messages qu'on peut déterminer une stratégie de priorité dans la gestion du traitement des messages.

CONVERSE fournit aussi les primitives pour l'équilibrage de charge dynamique des langage supérieurs comme CHARM++ ou AMPI. D'abord c'est CONVERSE, qui implémente un *seed balancer*, qui détermine où les nouveaux objets sont créer. Le *seed blancer* affecte donc les objets aux processeurs avec le désir d'équilibrer les charges dès le départ.
En plus, le support exécution implémente un système d'équilibrage de charge qui est basée sur des mesures. Pour cela, il place sur chaque processeurs un gestionnaire d'objet (*object manager*), qui collecte automatiquement les temps de calcul et les aptitudes de communication pendant l'exécution respectivement aux objets CHARM++ ou aux *user-level threads* d'AMPIdu processeur. Aussi, ces gestionnaires d'objet ont la capacité de créer ou de détruire les objets sur le processeur. Les données collectées sont alors passer à un contrôleur qui les enregistre dans une base de données appelé *load balancing database*. Cette base de données est distribué sur tous les processeur afin de minimiser les communication (voir figure 2.4).
Remarquons que le système d'équilibrage de charge se base ici sur le principe de persistance, c'est à dire il existe une corrélation temporelle des aptitudes de calcul et de communication, de telle sorte que l'on peut prédire le comportement futur d'une application par les mesures enregistrées dans le passé. Cela permet un équilibrage des charges effectif basé sur des mesures sans avoir la connaissance spécifique d'une application.
Ensuite, on dispose d'une collection de stratégies d'équilibrage des charges (*load balancing strategies*) qui détermine les nouvelles allocations des objets aux processeurs basés sur les informations fournis par le contrôleur de la base de données. Les développeurs de CONVERSE/CHARM++ proposent une série de stratégies avec différente propos. Notamment, on a le choix entre des stratégies centrales ou des stratégies distribuées. La stratégie centrale dispose d'une vue globale sur tout le système et base son équilibrage sur la construction d'un graphe d'objets complet. Notamment, ce graphe d'objets reprend la communication entre les objets ainsi que leurs charges CPU. Pour minimiser la communication, dans les stratégies distribuées le contrôleur de la base de données communique uniquement avec les proches voisins de son processeur et le graphe d'objets à la base est donc seulement partiel. Mais, un déséquilibre éventuel est donc distribué plus lentement qu'en stratégie centrale.

Quelques stratégies d'équilibrage sont :

- **Greedy :** C'est une stratégie centrale avec une nature agressive. Elle crée une liste d'objets décroissante selon leur consommation. Cette consommation d'un objet est déterminé avec le temps CPU de l'objet et le coût de communication (nombre de messages, taille des messages, ...) que ce objet a. Une deuxième liste contient les processeurs selon leur charge croissante. La charge d'un processeur est déterminée avec la somme des temps de calcul des objets qu'il détient. La stratégie *greedy* affecte alors l'objet le plus consommateur non encore affecté sur le processeur le moins chargé. Elle adapte ensuite les listes, et commence une nouvelle affectation selon le même schéma jusqu'au moment où tous les objets sont affectés. Comme cette

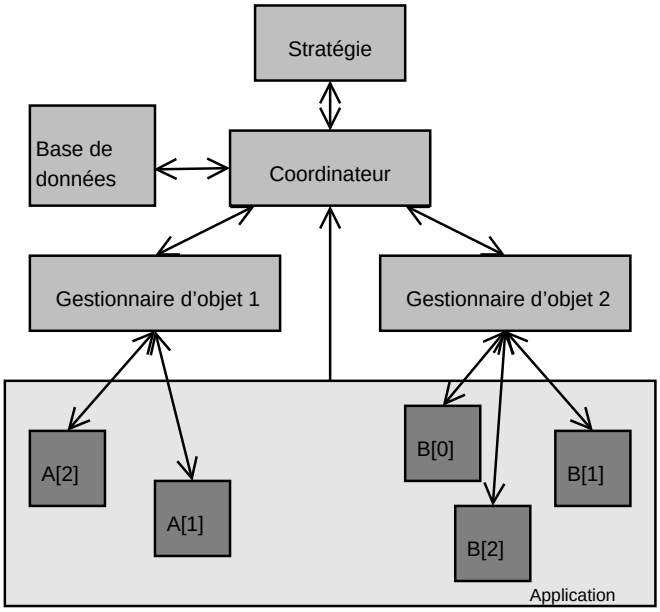


FIG. 2.4 – *Le système d'équilibrage de charge de* CONVERSE/CHARM++.

stratégie ne prend pas en compte l'affectation actuelle des objets, nous pouvons avoir beaucoup de migrations.

- **Refine :** Cette stratégie plus défensive est aussi centrale et essaye de limiter le nombre de migrations. Elle regarde uniquement les processeurs surchargés. Pour chaque processeur surchargé, cette stratégie migre alors objet par objet vers des processeurs moins chargés jusqu'au moment où la sur-charge du processeur est en-dessous d'un seuil tolérable.
- **Workstation :** Cette stratégie distribuée prend en compte au niveau local la charge réel d'un processeur et des processeurs voisins. Contrairement aux deux premières stratégies, la charge réelle d'un processeur comprend à coté de la charge due aux objets, la charge que d'autres tâches externes provoquent sur les processeurs. Ainsi, cette stratégie prend en compte des déséquilibres dûs à des charges externes et modifie l'affectation des objets pour équilibrer la charge totale.

Les trois stratégies présentées ici, ainsi que les autres stratégies proposées (voir [14]), sont toutes implémentées en modules séparés de l'implémentation CONVERSE. En incluant un de ces modules c'est l'utilisateur qui détermine à la compilation ou pendant l'exécution la stratégie à utilisée pour l'équilibrage des charges. Aussi, il est possible d'activer plusieurs stratégies pour une application et de changer ainsi de stratégie pendant l'exécution. Par exemple, on pourrait au début d'une exécution appliquer une stratégie agressive qui équilibrerait la charge grossièrement et puis, on passerait à une stratégie plus fine. De même, un utilisateur peut créer sa propre stratégie et l'implémenter dans un module afin de l'inclure dans le système d'équilibrage de charge de CONVERSE.

c) L'équilibrage de charge en CHARM++

Comme nous avons vu (page 38) les tableaux de *chares* constituent la structure de base pour des objets migrables [14]. En plus, la création d'un tableau incite le support exécution à collecter les informations sur ses éléments (*chares*) pour la base de données du contrôleur d'équilibrage de charge. Aussi, CONVERSE propose des API's pour l'équilibrage de charge

[14] En CHARM++ un simple objet *chare* est aussi migrable en appelant les API adaptées de CONVERSE. Cette approche complexe est cependant déconseillée par les développeurs de CHARM++.

des éléments du tableau. Par exemple, le programmeur peut individuellement déterminer si un *chare* d'un tableau peut être migrer ou non.
Selon les besoins de l'application, il existe en Charm++ trois méthodes pour utilisées l'équilibrage de charge. Ces méthodes déterminent quand et comment la phase de l'équilibrage des charges commence.

- **automatic without Sync :** Cette méthode peut lancer automatiquement et à tout moment la migration. Souvent on spécifie un intervalle `LBPeriod` après laquelle un test de migration est lancé. Par exemple, si `LBPeriod` vont 5 alors chaque cinquième seconde un test de migration est lancée.
- **automatic with Sync :** Dans cette méthode la migration est aussi lancée automatiquement, mais uniquement à des moments précis de l'exécution. Comme en AMPI tous les objets doivent appeler une primitive `atSync` qui demande un test de migration. Le test de migration et les migrations éventuelles auront lieu seulement si tous les objets ont invoqué cet appel. Remarquons que `atSync` est non-bloquant, mais l'envoi de message est postposé jusqu'à la fin de l'étape d'équilibrage de charge. Ce procédé est nécessaire car différentes formes de message de Charm++, par exemple un message collectif à un groupe de *chares*, peut rendre impossible une migration.
- **manual mode :** Dans ce monde, c'est au programmeur de programmer le début de l'équilibrage de charge. Ici c'est un objet qui lance avec `startLB` l'équilibrage de charge chez tous les objets. C'est alors au programmeur de vérifier que tous les objets sont migrables jusqu'à la fin de l'étape d'équilibrage de charge.

En résumé, un tableau avec nos exigences pour Charm++/Converse :

Exigences	**Généralité**	**High-level**	**Efficacité**	**Expressivité**	**Mod., Scal. et Réut.**
Charm++ Converse	applicable sur les machines MIMD	environnement parallèle avec un ensemble de fonctionnalité *high-level* complète	ensemble d'objet interagissant en *message-driven*	langage complexe	extension d'un code C++

2.6 Modèles basés sur le concept de tâche

Ce modèle utilise les concepts de plusieurs flux de contrôle. La notion de tâche est souvent implémentée via le principe de *thread* (voire aussi page 16). Ainsi, la coordination est effectuée via le partage de données entre *threads*. En général, ces modèles sont donc conçus pour des machines à mémoire partagée.

2.6.1 Modèles multi-threads *low-level*

La création et les interactions entre les *threads* sont ici explicites. Comme exemple, nous pouvons notamment citer Java.

Java

Java [25] [27] inclut la sémantique des *threads* dans son langage orienté-objet portable. Ainsi, on peut avec le principe du *multi-threading* créer des programmes qui exécutent plusieurs tâches en parallèle. Naturellement, à moins de posséder un ordinateur multi-processeurs, un programme Java s'exécute sur un seul processeur. Le *multi-threading* donne alors une illusion de parallélisme. Néanmoins, ce *multi-threading* sur un mono-processeur a

des avantages en terme de performance. Par exemple, semblable au principe de virtualisation en AMPI, quand un *thread* est bloqué par une opération d'entrée/sortie le processeur peut passer à un autre *thread*.

En Java, la gestion des *threads* et de la concurrence entre ces *threads* se fait de manière explicite. Un *thread* est créé en étendant la classe `Thread` ou en implémentant l'interface `Runnable`. Les passages entre états (bloqué, actif, exécutable,nouveau) d'un *thread* sont programmé explicitement. Par exemple, pour passer d'un *thread* à un autre on doit bloquer le *thread* actif en appelant une méthode `yield()`, `sleep()` ou `wait()`. La méthode `yield` permet d'exécuter un autre *thread* s'il en existe, `sleep` bloque le processus pour un certain intervalle de temps tandis que `wait()` met le *thread* dans un état bloqué duquel il peut seulement sortir si un autre *thread* appelle la méthode `notify()`. Aussi, on protège des méthodes critiques pour des accès concurrents à des données, par une étiquette `synchronized`, afin que la *Java Virtual Machine* n'exécute seulement qu'une méthode *synchronized* à la fois.
Comme en Java, nous n'avons pas de *user-level thread* la gestion des *thread*, spécialement le passage entre état, dépend aussi toujours de l'implémentation des *threads* du système d'exploitation sur lequel le programme Java s'exécute.

Remarquons, qu'il existe des possibilités de créer des applications distribuées. Une solution très *low-level* constitue la communication par "socket" où chaque communication doit être programmée au niveau des protocoles TCP ou UDP. Au contraire, des environnements *middleware high-level* comme le Java/RMI ou CORBA fournissent beaucoup de fonctionnalités avancées. Cependant, ces environnements ont un certain degré de complexité et, dû au haut coût de communication, le calcul parallèle par des tels environnements n'est pas très efficace. De plus, des environnements d'équilibrage de charge performants ne sont relativement peu développés pour ce domaine.

Exigences	**Généralité**	**High-level**	**Efficacité**	**Expressivité**	**Mod., Scal. et Réut.**
Java	applicable sur les machines SMP	programmation explicite des *threads*	faible efficacité de la *Java Virtual Machine*	concept explicite de *multi-threading*	langage orientné-objet portable

2.6.2 Modèles multi-threads *high-level*

En complément, au contrôle explicite des threads quelques environnements fournissent des constructions plus élevées.

A) PM2 (Parallel Multithreaded Machine)

PM2 [37] est un support d'exécution qui est développé l'université de Lille et à l'ENS Lyon. L'objectif de PM2 est de fournir un environnement portable pour des application *multi-threading* fortement irrégulières sur des architectures à mémoire partagée mais aussi sur des machines à mémoire distribuée.
PM2 fourni une interface semblable au POSIX-*threads* pour la création, la gestion et la synchronisation des *threads* dans des contextes distribués. Le mécanisme de base pour la communication est le RPC (*Remote Procedure Call*). En utilisant de tels appels distants, les *threads* de PM2 peuvent invoquer l'exécution distante de services définis par l'utilisateur. Un appel distant peut être traité par un *thread* existant ou peut provoquer la création d'un nouveau *thread*. Aussi, des *threads* peuvent se partager une mémoire ou en utilisant des RPC transférer des données entre mémoire distribuée.

Le support d'exécution de PM2, appelé μPM2, contient deux composants principaux (voir figure 2.5). Ainsi, pour le *multi-threading* il utilise Marcel qui est une bibliothèque efficace pour des *threads* semblable aux POSIX-*threads*. Et pour assurer la portabilité au niveau réseau, la bibliothèque Madeleine, qui supporte efficacement une large gamme d'interfaces de communication, comme TCP ou MPI, mais aussi des protocoles spécialisée de haute-performance comme BIP [15]. Remarquons, que c'est donc Madeleine qui implémente le système RPC, qui est en réalité basé sur des envois et des réceptions de message, effectués par des *threads*[15] dédié à la communication.

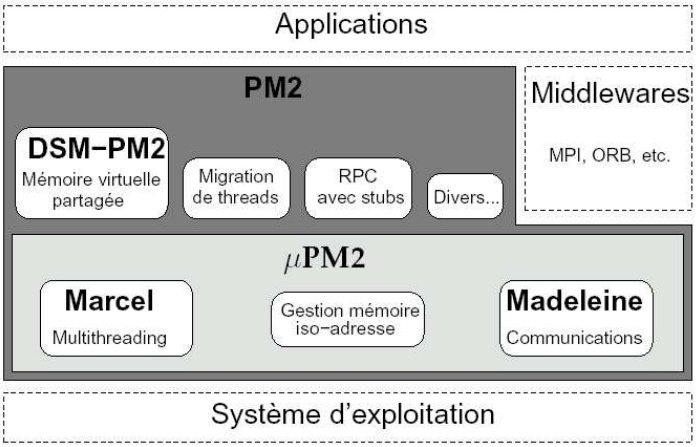


FIG. 2.5 – *L'architecture PM2* (source [37]).

Marcel utilise un ordonnanceur qui permet une affectation flexible et efficace des *threads* à la fois d'une manière globale ou local. Cet ordonnanceur mixte ou ordonnanceur à deux niveaux s'applique aux *threads* système exploitation ainsi qu'aux *user-level threads*. L'ordonnanceur affecte alors sur chaque processeur des *threads* noyau (niveau système exploitation) qui vont alors faire la gestion d'ordonnancement au niveau des *user-level threads* locaux. L'approche globale est caractérisée par une file d'attente centrale des *threads* à traiter. Dans l'approche locale, chaque *thread* noyau a sa file d'attente locale. Dans ce deuxième cas, PM2 instaure alors un mécanisme de *workstealing*(vol de *threads* chez d'autres processeurs) pour veiller à ce que chaque *thread* noyau soit occupé.
Pour notre démarche une propriété intéressante de PM2 est son mécanisme de migration de *threads* qui permet de déplacer des *threads* en transparence pendant leur exécution d'un processeur à un autre. Comme CHARM++, PM2 permet ainsi de définir indépendamment de l'application utilisateur des stratégies d'équilibrage dynamique de charge. Le mécanisme de base pour les migrations est une approche *iso-adresse*. Cette approche inventée par les développeurs de PM2, a été déjà présentée dans la description de CONVERSE sous le terme iso-adresse, et garanti donc que l'espace de mémoire virtuelle allouée par une tâche sur un processeur est laissé libre sur un autre processeur. Ainsi, il est possible de migrer un *thread*, simplement en copiant ses données à la même adresse que sur le processeur initial. Finalement, les développeurs de PM2 ont encore développé un module spécial pour la mémoire virtuellement partagée afin de pouvoir gérer des migrations de *threads* qui se partagent des données. Ainsi, le DSM-PM2 est une plate-forme générique pour l'implémentation de protocoles de cohérence pour systèmes à mémoire virtuellement partagée.

[15]Dans leurs sémantique ces *threads* sont semblable aux méthodes d'entrée des *chares* de CHARM++.

PM2 fournit aussi un langage propre qui est une extension du langage C. Ce langage implémente le modèle SPMD classique et contient des primitives classiques comme l'initialisation, qui lance sur tous les processeurs disponibles une copie du programme, et la fermeture de l'environnement PM2, des primitives pour la gestion des services et/ou *thread* qui peuvent être appelés via le mécanisme RPC ou encore des primitives pour la gestion de migration. Un simple exemple d'un "hello World" pour PM2 est donnée si après :

```
/*Programme helloPM2*/

#include <stdio.h>
#include <pm2.h>

int pm2_main(int argc, char *argv[]){

  pm2_init(&argc, argv);

  tprintf("Hello World!\n");

  if (pm2_self() == 0)
     pm2_halt();

  pm2_exit();
  return 0;
}
```

Cependant, les développeurs de PM2 mettent plus d'efforts de recherche sur le support exécution PM2 que sur le langage PM2 lui-même. En effet, PM2 peut fournir des services *high-level* via ses interfaces pour des applications écrites dans des contextes Java ou HPF . Remarquons pour le lecteur intéressé, que l'environnement Nexus[39] fournit un support semblable.

Exigences	**Généralité**	**High-level**	**Efficacité**	**Expressivité**	**Mod., Scal. et Réut.**
PM2	applicable aux machine MIMD-SM et MIMD-DM	équilibrage dynamique de charge avec *iso-adresse*	avantage du multi-threading	abstraction de message-passing par RPC	environnement portable

B) Cilk

Cilk [17] [32] est une extension du langage C implémentant le principe de *multi-threading* qui a été développé au MIT à lafin des années nonante. Comme en CHARM++ la philosophie de Cilk est *high-level*. En Cilk le programmeur se concentre sur la structuration de son programme parallèle en laissant au support exécution l'ordonnancement des tâches et l'exécution efficace sur une architecture donnée. Ainsi, le support d'exécution de Cilk prend en charge l'équilibrage de charge et les protocoles de communication. Spécialement, l'environnement Cilk garantit une efficacité en donnant des outils pour prédire la performance.
L'environnement Cilk a été spécialement développé pour des applications dynamiques et irrégulières pour machines SMP. Le parallélisme de tâche est obtenu en indiquant à l'environnement Cilk des fonctions qui peuvent être calculées simultanément. Pour cela, regardons le programme Cilk suivant qui calcule le n^{ime} nombre Fibonacci [16] :

[16] $\begin{cases} F(0) = 0, F(1) = 1 \\ F(n) = F(n-1) + F(n-2) \end{cases}$

```
/*Programme fibCilk*/

#include <stdlib.h>
#include <stdio.h>
#include <cilk.h>

cilk int fib(int n){
  if (n<2)
     return n;
  else {
     int x, y;

     x = spawn fib (n-1);
     y = spawn fib (n-2);

     sync;

     return (x+y);
  }
}

cilk int main(int argc, char *argv[]){
  int n, result;

  n = atoi(argv[1]);
  result = spawn fib(n);

  sync;

  printf("Result: %d\n", result);
  return 0;
}
```

La seule différence avec un programme C est inclusion de la bibliothèque "*cilk.h*" et l'utilisation des trois mots clés `cilk`, `spawn` et `sync`.
Le mot clé `cilk` identifie une procédure Cilk, qui est en fait une version parallèle d'une fonction C classique. La plupart du code est exécutée en séquentiel. Uniquement l'appel d'une fonction avec le mot clée `spawn` va introduire du parallélisme. Dans le programme exemple précédent, l'appel à `spawn fib()` provoque l'exécution du code de la fonction fib dans un *thread* fils. C'est alors le support d'exécution qui décide de l'ordonnancement donc par exemple du lieu d'exécution de ce fils. Un père peut créer plusieurs fils afin d'augmenter le calcul effectué en parallèle.
Cependant un père ne peut pas utiliser en sécurité les résultats de ses fils avant un appel à `sync`. Au contraire des synchronisations globales de MPI, `sync` est une barrière locale, qui retourne dès que tous les fils du père ont terminé.
En plus, Cilk permet d'utiliser la mémoire partagée entre plusieurs *threads*. Pour ce propos, Cilk met à disposition des fonctions de contrôle d'accès comme *lock* et *unlock*. Ainsi, un *thread* peut accéder en sécurité à des sections critiques, donc à des données qui pouvaient être incohérentes dues à des accès parallèles incontrôlés.

Avec la suite de `spawn` et de `sync` l'environnement Cilk peut établir un arbre de tâches (figure 2.6). Dans cette arbre l'environnement identifie le chemin critique, c'est à dire la plus longue suite de tâches dans l'arbre. A partir de ce chemin, l'environnement est alors capable de déduire une performance estimée du programme.

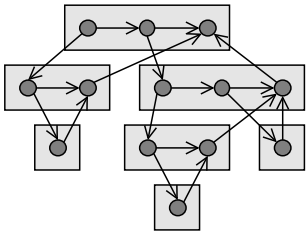


FIG. 2.6 – *Un arbre de tâches. Chaque rond représente un thread où une flèche horizontale indique un thread successeur (suite du père), une flèche vers le bas un spawn (thread fils) et une flèche vers le haut le retour d'un résultat (sync).*

L'ordonnanceur distribué de Cilk se base sur le principe de *workstealing*. C'est à dire, si un processeur de l'architecture devient libre il tente de "voler" une tâche d'un autre processeur. L'autre processeur est choisi au "hasard" par l'ordonnanceur qui analyse l'arbre de tâche afin de trouver un nouveau *thread* qui peut être exécuté sur le processeur libre.
Notons encore, Cilk est un vrai langage avec son propre compilateur. Ainsi, il ne peut pas utiliser les compilateurs standards et il souffre donc des mêmes problèmes d'optimisation de code que HPF.

Exigences	**Généralité**	**High-level**	**Efficacité**	**Expressivité**	**Mod., Scal. et Réut.**
Cilk	applicable sur les machines SMP	équilibrage de charge avec ordonnancement automatique	faible optimisation du code dû au compilateur propre	parallélisme de tâche (spawn)	programme C en ajoutant une dizaine de mot de clés

C) Athapascan

Athapascan est développé à l'université de Grenoble. Une première version, Athapascan-0 [2], à été développé début des années nonante, et était une extension du langage C. Athapascan-0 était programmé pour des machines SMP ou des clusters de SMP. Sa bibliothèque prend en charge la communication qui se base sur la gestion de *threads* (inter SMP) ou sur des communications MPI au cas des grappes SMP (intra SMP).

Basé sur l'environnemnt Athapascan-0, les développeurs de Grenoble ont sorti fin des années nonantes une deuxième version : Athapascan-1 [46]. L'environnement Athapascan-1 s'applique pour des applications irrégulières et fournit des fonctionnalités d'équilibrage de charge. Comme la première version, Athapascan-1 est un langage *multi-thread* avec une gestion de communication[17] asynchrone et fournit un support pour C mais aussi pour C++.
Pour expliquer le fonctionnement d'Athapascan-1 partons d'un exemple C++ qui calcule le n^{ime} nombre Fibonacci :

```
/*Programme fibAthapascan*/

#include <athapascan-1.h>
#include <iostream.h>
#include <stdlib.h>

struct add {
```

[17]Maintenant en Athapascan-1, la bibliothèque de communication fourni par Athapascan-0 était remplacée par une bibliothèque moins dépendant de l'architecture *Inuktitut* (Voir `http://www-id.imag.fr/Logiciels/inuktitut/`).

```
 void operator()(a1::Shared_r<int> a, a1::Shared_r<int> b, a1::Shared_w<int> c){
     c.write(new int(a.read()+b.read()));
 }
};

struct fibonacci {
 void operator()(int n, a1::Shared_w<int> res){
     if ( n < 2 )
        res.write( new int(n) );
     else {
        a1::Shared<int> res1 = int(0);
        a1::Shared<int> res2 = int(0);
        a1::Fork<fibonacci>()(n-1, res1);
        a1::Fork<fibonacci>()(n-2, res2);
        a1::Fork<add>()(res1, res2, res);
     }
 }
};

struct print {
 void operator()(int n, a1::Shared_r<int> res){
     cout << "Fibonacci(" << n << ")= " << res.read() << endl;
  }
};

int main(int argc, char *argv[]){
  a1::Community com = a1::System::create_initial_community(argc, argv);

  a1::Shared<int> res = int(0);
  a1::Fork<fibonacci>()(atoi(argv[1]), res);
  a1::Fork<print>()(atoi(argv[1]), res);

  com.leave();
  return 0;
}
```

Au début du main nous initialisons l'environnement Athapascan en créant un communicateur `com`. Puis, nous remarquons, pour la déclaration des variables, qu'on peut spécifier les dépendances entre données (ici partagée `shared` et leur mode d'accès (en lecture : `r`, en écriture : `w`). Ce procédé s'appelle typage des accès mémoire. La création du parallélisme des tâches se fait en appelant des `fork`, qui vont crée des *threads* qui exécutent l'opérateur `operator`, d'une structure `struct` défini au préalable.
Comme en Cilk, le modèle d'exécution d'Athapascan se base sur la création d'un graphe dynamique de tâches. Mais, Athapascan inclut dans ce graphe le flot de données (voir figure 2.7), qui n'était pas modélisé en Cilk. Avec la spécification de flot de données par le typage des accès mémoire l'environnement fournit une synchronisation implicite entre les *threads*. Ainsi, la sémantique de l'exécution correspond intuitivement à l'exécution du programme en séquentiel. En effet, avec une gestion des versions de données (succession des valeurs d'une donnée) chaque lecture d'une donnée voit la "dernière" écriture comme dans un ordre séquentiel d'exécution des *threads*. Aussi, l'exécution du programme est représenté dynamiquement dans l'arbre de tâches avec un principe d'évolution des états. Dans cette évolution un état (*thread* ou version de donnée) passe successivement de l'état *Attente* par l'état *Prêt* et l'état *Exécution* à l'état *Terminé*. Avec ces mécanismes l'environnement Athapascan-1 peut détecter le parallélisme et contrôle la sémantique du programme. Comme en Cilk, Athapascan fourni aussi un modèle de coût basé sur le graphe associé à un programme.

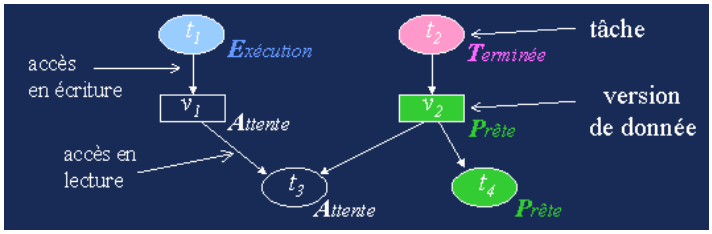


FIG. 2.7 – *Le graphe de tâche avec le flux de données et l'évolution des états en Athapascan*

La création et la communication avec des *threads* sur des processeurs distants se fait de manière automatique. L'environnement Athapascan crée des graphes de tâche partiels avec des réplications des versions de données sur chaque processeur. Il existe alors un lien spécial entre les versions répliquées qui est géré par l'environnement afin de garder l'intégrité des données. L'avantage est que les versions de données consultées par les tâches sont locales.
Pour la migration d'un *thread* l'environnement déplace la tâche vers un nouveau processeur et réplique la version des données de la tâche sur ce nouveau processeur. Afin de pouvoir répliquer les données, l'utilisateur doit spécifier des méthodes de sérialisation (pack et unpack). Avec les liens spéciaux entre versions répliquées la communication est assurée avec les tâches restées sur l'ancien processeur.
Comme le graphe de tâche tient compte de la dépendance des données, Athapascan-1 peut équilibrer les *threads* en respectant le principe de localité des données. C'est à dire, des *threads* qui accèdent aux mêmes données sont placées sur le même processeur afin de minimiser les coûts de communication. En cas de maque de travail, un processeur peut voler une série de *threads* "coopératif" (principe de *workstealing*). Cette solution distribuée est choisie afin d'éliminer le sur-coût d'une gestion globale.

Exigences	**Généralité**	**High-level**	**Efficacité**	**Expressivité**	**Mod., Scal. et Réut.**
Athapascan-1	applicable sur les machines SMP mais aussi sur des machines distribuées	équilibrage de charge	gestion avec arbre de tâche et utilisation de compilateur standard	deux primitives principales fork et shared	programme C et C++ étendue

D) OpenMP

L'objectif d'OpenMP [40] est d'utiliser au maximum les avantages d'un architecture à mémoire partagée. Semblable à HPF, OpenMP est basé sur une programmation par directives. En effet, en OpenMP on ajoute des commentaires au programme pour indiquer au compilateur OpenMp des régions parallèles du programme (voir figure 2.8). OpenMP est un standard qui est supporté par beaucoup de constructeurs de machines SMP [18]. Le standard OpenMP est défini pour des langages Fortran, C et C++.

Le principe d'OpenMP est un "*fork-join*" parallélisme. Tous les programmes commencent en séquentiel avec un *thread* maître. Ce programme séquentiel peut alors passer dans des régions parallèles ou le *thread* maître initial est étendue (fork ou spawn) par un

[18] Voir `http://www.openmp.org`.

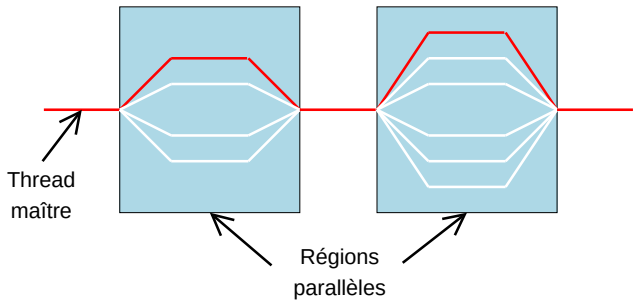


FIG. 2.8 – *Principe d'exécution d'un programme OpenMP avec un thread maître et différentes régions parallèles.*

groupe de *threads* selon le besoin. Après une sorte de synchronisation (join) on retrouve uniquement le *thread* maître qui s'exécute alors de nouveau en séquentiel.

Le standard OpenMP définit des directives qui vont indiquer au compilateur OpenMP des régions parallèles. Pour la création des *threads* donc le début de la zone parallèle pour un programme C on indique :

```
#pragma omp parallel
{
    code exécuté par chaque thread
}
```

Un mécanisme de base de ces régions parallèles est alors le partage du travail. Comme en HPF, des boucles peuvent être exécutées en parallèle en spécifiant une directive devant l'appel de boucle. Ainsi on peut profiter d'un parallélisme de données :

```
#pragma omp for
shared(a) private(i)
for (i=0 ; i<100 ;i++ ) {
}
```

Par exemple, si on travaille avec 4 *threads*, le premier *thread* exécute la boucle avec `i` de 0 à 25, le deuxième de 26 à 50 et ainsi de suite. Les directives `shared` et `private` indiquent si les variables sont partagées par les *threads* ou privées à un *thread*. Aussi, la notion de section permet d'affecter à différentes *threads* différentes parties d'un code :

```
#pragma omp parallel
{
   #pragma omp section
   {
       job1() ;
   }
   #pragma omp section
   {
       job2() ;
   }
}
```

Vu qu'on peut partager des données, il est parfois nécessaire de contrôler l'accès aux données afin de préserver leur intégrité. Ainsi, il existe des primitives qui indiquent des sections critiques et qui fournissent une exclusion mutuelle (`#pragma omp critical(left)`) ou qui exécute une section de code en atomique, c'est à dire comme si c'était une instruction machine unique (`#pragma omp atomic`). En plus, il existe des barrières afin de

synchroniser tous les *threads* (`#pragma omp barrier`), ou de spécifier le code dans la région parallèle qui est uniquement exécuté par le *thread* maître (`#pragma omp master`).

Finalement, en plus des directives, OpenMP fournit aussi quelques fonctions dans une bibliothèque "omp.h". Par exemple, la fonction `omp_get_thread_num()` peut être utilisée pour déterminer l'identificateur du *thread* en cours d'exécution. Ainsi, à coté du parallélisme des données le modèle SPMD peut être implémenté en OpenMP. Aussi, OpenMP prend en compte des variables d'environnement du système exploitation afin de déterminer par exemple le nombre de *threads* maximal lancés pour une région parallèle.

Exigences	**Généralité**	**High-level**	**Efficacité**	**Expressivité**	**Mod., Scal. et Réut.**
OpenMP	standard pour les machines SMP	pas d'équilibrage de charge	les constructeurs fournissent des compilateurs adaptés à l'architecture SMP	parallélisme par directive semblable au HPF	programmes séquentiels modifiés (directives et appels système)

2.7 Tableau récapitulatif

Afin, de pouvoir mieux comparer les différentes environnements dans le chapitre suivant, nous reprenons le tableau des exigences de tous les environnements discutés :

Exigences	**Généralité**	**High-level**	**Efficacité**	**Expressivité**	**Mod., Scal. et Réut.**
HPF	uniquement parallélisme des données	pas d'équilibrage de charge et très difficile pour des applications irrégulières	pas de compilateurs optimisés	pas de parallélisme vraiment explicite	code séquentiel Fortran avec directives
PVM	applicable sur les machines MIMD	gestion dynamique limitée des ressources	utilisation des compilateurs standards optimisés	parallélisme explicite avec lancement de processus (spawn)	programmes séquentiels modifiés (appels système et gestion de la machine virtuelle parallèle)
MPI	standard applicable sur toutes les machines MIMD	pas d'équilibrage de charge et aucune gestion dynamique possible	support des constructeurs et possibilité d'optimisation des communications via définition de topologie	parallélisme explicite	programmes séquentiels modifiés (appels système)
AMPI	applicable sur les machines MIMD	équilibrage dynamique de charge	virtualisation des processeurs	parallélisme explicite	code MPI avec adaptation pour la migration

Exigences	Généralité	High-level	Efficacité	Expressivité	Mod., Scal. et Réut.
Charm++ Converse	applicable sur les machines MIMD	environnement parallèle avec un ensemble de fonctionnalité *high-level* complète	ensemble d'objet interagissant en *message-driven*	langage complexe	extension d'un code C++
Java	applicable sur les machines SMP	programmation explicite des *threads*	faible efficacité de la *Java Virtual Machine*	concept explicite de *multi-threading*	langage orientné-objet portable
PM2	applicable aux machine MIMD-SM et MIMD-DM	équilibrage dynamique de charge avec *iso-adresse*	avantage du multi-threading	abstraction de message-passing par RPC	environnement portable
Cilk	applicable sur les machines SMP	équilibrage de charge avec ordonnancement automatique	faible optimisation du code dû au compilateur propre	parallélisme de tâche (spawn)	programme C en ajoutant une dizaine de mot de clés
Athapascan-1	applicable sur les machines SMP mais aussi sur des machines distribuées	équilibrage de charge	gestion avec arbre de tâche et utilisation de compilateur standard	deux primitives principales fork et shared	programme C et C++ étendue
OpenMP	standard pour les machines SMP	pas d'équilibrage de charge	les constructeurs fournissent des compilateurs adaptés à l'architecture SMP	parallélisme par directive semblable au HPF	programmes séquentiels modifiés (directives et appels système)

Tab. 2.1 : *Tableau de comparaison des environnements discutés selon les exigences d'un système parallèle.*

Chapitre 3

Choix technologiques

Ce chapitre a pour rôle de faire la liaison entre les deux premiers chapitres théoriques et la suite de ce document qui va traiter la problématique de ce mémoire d'un côté plus pratique. A présent, nous allons faire une comparaison du point de vue académique sur les différents environnements discutés dans le chapitre précédent.

Comme point de départ, nous pouvons prendre le tableau récapitulatif 2.1. Selon les objectifs fixés au début : à savoir un système *high-level* avec équilibrage dynamique de charge, appliquant les conventions communes en calcul parallèle comme le standard MPI et utilisable sur un cluster de PCs ; nous allons alors citer les avantages et désavantages des différents environnements.

3.1 Choix des systèmes *high-level*

Selon les exigences de la section 2.1 un système *high-level* cherche la division optimale du travail de programmation entre le système (par exemple un support d'exécution) et le programmeur. En effet, des systèmes *low-level* laissent tout le travail au programmeur. Par exemple, le standard MPI ne spécifie rien sur l'ordonnancement, l'affectation des tâches de calcul au processeur, d'un système d'équilibrage de charge ou encore de la décomposition d'un programme afin de pouvoir le calculer en parallèle. Au contraire, les systèmes *high-level* cherchent à automatiser une ou plusieurs de ces tâches de programmation parallèle. En plus, à ce degré d'automatisation des différentes systèmes *high-level*, nous pouvons encore définir selon une deuxième axe un niveau de spécialisation dans les tâches automatisées (voir figure 3.1). Par exemple, des systèmes *high-level* qui fournissent l'équilibrage de charge peuvent être comparer en terme d'équilibrage statique, d'équilibrage dynamique ou encore selon les stratégies de migration proposées.
Pour le degré d'automatisation, la plupart des systèmes *high-level* actuels se trouvent au niveau d'équilibrage de charge. Par exemple, Charm++ se trouve à ce niveau. L'automatisation de la décomposition, donc la parallélisation automatique d'un programme séquentiel, est laissée au programmeur, car celui-ci à une meilleure connaissance de la nature de l'application. Aussi, les approches de parallélisation automatique rencontrent encore beaucoup de problèmes, notamment au niveau de l'efficacité du code parallèle produit. Le niveau de spécialisation différencie fortement selon les applications cibles des systèmes *high-level*.
Selon nos objectifs, nous évaluerons alors les différents environnements *high-level* de notre état de l'art selon leur capacité d'équilibrage dynamique de charge et leurs stratégies de migration proposées.

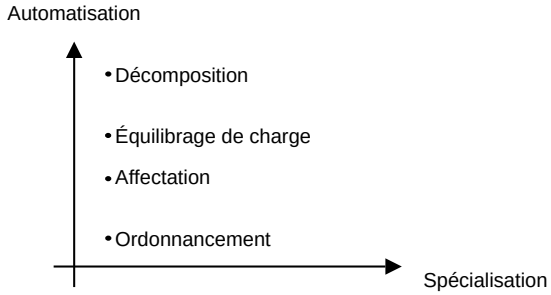


FIG. 3.1 – *Les systèmes high-level peuvent être classés selon leur degré d'automatisation des tâches dans la programmation parallèle et selon le niveau de spécialisation dans les différentes tâches automatisées.*

3.2 Choix de l'architecture cible et du standard de calcul parallèle

Comme architecture cible nous avons spécifié dans les objectifs un cluster de PC Linux. En effet, les architectures clusters trouvent de plus en plus de succès dans des entreprises et institutions [1]. Dans le domaine du calcul parallèle, les clusters sont une solution idéale en terme de performance et de coût par rapport aux ordinateurs parallèles classiques. Par exemple, les tests pratiques des deux chapitres suivants ont été effectués sur un simple réseau de PC. Le choix du système d'exploitation Linux résulte du fait que la plupart des environnements parallèles sont développés dans des contextes UNIX. Remarquons, que tous les environnements présentés sont *open source*.
Notre évaluation regarde donc si les environnements sont applicables dans des architectures clusters. Spécialement, le support de mémoire distribuée, habituel dans un cluster, reçoit une attention particulière.

Notre volonté d'utiliser un standard de calcul parallèle est soutenu par un objectif de réutilisabilité. Ainsi, beaucoup d'applications existantes en calcul parallèle ont été écrites avec des bibliothèques traditionnelles. Il serait donc intéressant que notre environnement supporte ces concepts connus, afin de servir des services *high-level* à ces applications déjà efficaces. Le message-passing de MPI est un tel standard largement connu et utilisé en calcul parallèle. En plus, il est facilement applicable sur un cluster de PC. Néanmoins, certains problèmes irréguliers peuvent considérablement augmenter leurs performances avec des environnements *high-level* [12].

3.3 Évaluation des environnements

A partir du tableau des exigences 2.1 et selon nos objectifs nous allons lister très schématiquement les avantages (+) et les désavantages (−) de chacun des environnements *high-level*.

3.3.1 Charm++

+ + + **Équilibrage dynamique :** CHARM++ avec l'aide de CONVERSE met à disposition l'équilibrage dynamique de charge selon différentes stratégies centrales et distribuées.

[1] Voir aussi `http://www.top500.org/`.

Il est même possible de récrire l'ordonnanceur si le programmeur dispose des informations spécifiques de l'application qui permettent un ordonnancement plus efficace.

++ **Efficacité :** Vu que CHARM++ est *message-driven* et basé sur le concept orienté-objet, il comble le temps d'attente d'une communication par le traitement d'un nouveau message disponible. Avec le principe de virtualisation, donc en plaçant un nombre important de processus sur un processeur, l'utilisation des processeurs est augmentée. CHARM++ a reçu différents prix (Bell Award) pour des applications dans le domaine du dynamique moléculaire.

\+ **Fonctionnalités *high-level* :** Un logiciel de visualisation des performances *Projection* des programmes CHARM++ et un framework *Faucets* pour l'interconnexion et l'ordonnancement entre différents clusters sont fournis avec l'environnement. De plus, des recherches actuelles sur le *checkpointing* devraient fournir une gestion de la tolérance aux pannes et faciliter la migration.

\+ **Portabilité :** Grâce à la portabilité de CONVERSE l'environnement CHARM++ est disponible pour différents systèmes d'exploitation(linux, windows) et pour différents modes de communications(Ethernet, Myrinet, MPI). Ainsi CHARM++ est utilisable sur des systèmes à mémoire distribuée ou sur des systèmes à mémoire partagé.

\− **Complexité :** Le langage CHARM++ supporte beaucoup de fonctionnalités ainsi que des adaptations par le programmeur. Néanmoins, ce grand ensemble de possibilités apparaît complexe et diminue l'expressivité à cause des appels parfois *low-level*.

−− **Reutilisabilité :** Il est très difficile de réutiliser un programme séquentiel ou parallèle en CHARM++. Souvent le code doit être complètement récrit. Les concepts traditionnels de *message-passing* ne sont pas disponibles.

3.3.2 AMPI

+++ **Reutilisabilité et Équilibrage dynamique :** AMPI permet d'utiliser des programmes classiques MPI avec presque toutes les fonctionnalités *high-level* de CHARM++ ou CONVERSE. Ainsi, on a des programmes MPI sur lesquels un équilibrage dynamique de charge est possible.

++ **Simplicité :** La transformation de programme MPI en AMPI nécessite de petites transformations, notamment la migration demande quelques appels supplémentaires. Pour le parallélisme, nous avons ainsi la même expressivité qu'un programme MPI.

++ **Efficacité :** Avec le principe de virtualisation où plusieurs processus MPI peuvent être placés comme *user-level threads* sur un processeur réel l'environnement AMPI peut combler les attentes dues aux appels bloquants de MPI. Aussi, des applications MPI irrégulières ou des exécutions sur des architectures hétérogènes peuvent augmenter de performance due aux capacités d'équilibrage de charge d'AMPI.

\+ **Portabilité :** Comme AMPI et CHARM++ se partagent le même support d'exécution CONVERSE ils ont aussi la même portabilité. Ainsi, AMPIest utilisable sur des architectures à mémoires partagée ou distribuée.

\− **Sur-coût migration :** Pour des applications statiques ou régulières, c'est à dire très prévisibles, un programme AMPI peut être moins efficace qu'un programme MPI dû au sur-coût de la gestion des migrations.

3.3.3 PM2

+++ **Équilibrage dynamique :** PM2 se base sur le principe de *multi-threading* et avec le mécanisme *iso-adresse* ainsi que des stratégies de migration un déplacement d'un

thread est facile. Aussi, PM2 fournit un ordonnanceur mixte qui permet une allocation des *threads* à un niveau global et local.

++ **Portabilité :** Le support d'exécution PM2 est conçu en différents modules et envisage comme objectif d'être portable. Ainsi, PM2 est utilisable sur une large gamme d'architecture avec mémoire distribuée et mémoire partagée. Un module prend spécialement en charge la mémoire virtuellement partagée.

\+ **Efficacité :** La bibliothèque de communication prend en charge différentes interfaces de communication. PM2 peut donc utiliser la communication la plus performante selon l'architecture sous-jascente pour implémenter le RPC de PM2.

−− **Utilisation et Reutilisabilité :** Le langage de PM2 est peu utilisé et les programmes séquentiels ou parallèles existants doivent être reimplémentés si on veut utilisé le langage PM2 spécifique. Mais, le projet PM2 constitue un bel effort pour le développement d'un support d'exécution plus complet que CONVERSE.

3.3.4 Cilk

++ **Équilibrage de charge :** Cilk fournit un ordonnanceur distribué qui applique le principe de *workstealing* arbitraire[2] afin de garantir une bonne utilisation du système. C'est aussi le système qui va s'occuper d'allocation des *threads* fils créés sur des processeurs.

\+ **Simplicité et Reutilisabilité :** Les primitives Cilk sont peu nombreuses et permettent au programmeur d'indiquer la structuration et décomposition du programme parallèle. C'est le système qui se charge des autres fonctionnalités *high-level*. Un programme séquentiel en C est avec un peu d'effort transformable en Cilk. Par contre, cette possibilité est inexistante pour des programmes parallèles existants.

\+ **Prédictabilité :** Avec la construction d'un arbre de tâches, Cilk fournit un modèle de coût et des outils afin de pouvoir prédire et garantir l'efficacité du programme parallèle sur une architecture donnée.

\− **Efficacité :** Cilk utilise son propre compilateur est ne peut donc pas profiter des optimisations que les compilateurs standards fournissent. Aussi, l'arbre de tâches et l'ordonnanceur ne prennent pas en compte des coûts de communication.

−− **Portabilité :** Cilk est développé pour des applications irrégulières sur des machines à mémoire partagée. Ainsi, il ne possède pas des mécanismes de migration pour les états des *threads*.

3.3.5 Athapascan-1

++ **Équilibrage de charge :** Athapascan-1 fournit un équilibrage de charge basé sur son arbre de tâche. La migration des *threads* est réalisée avec des réplications des versions de données (voir page 50) chez le nouveau processeur. L'utilisation du *workstealing* garantit une meilleur utilisation des ressources.

++ **Efficacité et Prédictabilité :** L'environnement Athapascan-1 utilise des compilateurs standards (suite de gnu) et se base sur une bibliothèque de communication évoluée entre *threads*. Au contraire de Cilk, l'arbre de tâche d'Athapascan-1 prend en charge les flots de données entre tâches et permet ainsi une optimisation dans la gestion des communications et une meilleure prévision et garantie de la performance du programme.

[2] C'est à dire on vole du travail sur un processeur choisi au hasard.

+ **Portabilité :** Des architectures à mémoire partagée et distribuée sont supportées par l'environnement.

+ **Expressivité :** Un point de vue intéressant dans Athapascan-1 est la possibilité du typage de l'accès mémoire. Ainsi on peut exprimer à la fois le parallélisme des tâches et le parallélisme des données.

−− **Réutilisabilité :** La transformation d'un programme séquentiel en Athapascan-1 est difficile et la conversion d'un programme parallèle traditionnel est impossible.

3.3.6 OpenMP

++ **Efficacité et Utilisation :** OpenMP est un standard respecté par les constructeurs des machines SMP. Sur ces architectures il est considéré comme très efficace et est beaucoup utilisé en industrie.

++ **Réutilisabilité et Expressivité :** Des programmes séquentiels sont adaptables pour OpenMP en introduisant des directives qui spécifient des régions parallèles. Par contre, la conversion d'un programme parallèle est peu supportée à l'exception des programmes HPF et du modèle du parallélisme de données.

−− **Portabilité :** OpenMP ne supporte pas les machines à mémoire distribuée.

− − − **Équilibrage de charge :** L'environnement OpenMP ne fournit aucune gestion d'équilibrage de charge.

3.4 Choix d'AMPI

Rappelons qu'une évaluation plus approfondie de l'environnement AMPI constituait un but de départ à ce mémoire. En effet, cet environnement récent[3] montre des caractéristiques intéressantes par rapport aux objectifs posés. Ainsi, ses capacités d'équilibrage dynamique de charge avec ses stratégies de migration multiples valent la peine d'une investigation plus profonde. Aussi, l'objectif d'utiliser des programmes parallèles traditionnels de MPI sur un environnement *high-level* sans transformation majeure est vérifié par AMPI. Un grand défi d'AMPI est sûrement aussi la réalisation efficace et performante d'une telle application parallèle sur une architecture cluster à mémoire distribuée.
Donc de tous les environnements discutés dans ce mémoire AMPI fournit, au moins théoriquement, la meilleure réponse aux objectifs posés. Le défi des deux derniers chapitres est alors de vérifier concrètement si l'environnement AMPI peut satisfaire tous ces objectifs posés.

[3]Comme on va le voir, le développement d'AMPI n'est pas encore clôturée.

Chapitre 4

Tests d'ordonnancements - Le programme *sum*

Ce chapitre a pour objectif de vérifier les déclarations faites par les développeurs AMPI[1]. Avec une simple application test, le programme *sum*[2], nous allons faire une série de tests qui montrent les avantages d'un environnement dynamique et *high-level* AMPI par rapport au standard statique MPI.
Le chapitre suivant vérifiera alors si les résultats établis dans ce cadre de test sont encore d'actualité dans un cadre réel.

4.1 Exigences et utilités de notre programme test

Afin de vérifier les performances de l'environnement AMPI, notre programme test doit pouvoir répondre aux exigences suivantes :

- **Équilibrage dynamique de charge :** Notre programme doit être capable de tester l'équilibrage dynamique de charge et les stratégies de migration. Afin, de pouvoir évaluer le système d'équilibrage, nous allons nous placer dans différents contextes.
  Ainsi, notre application test doit fournir une exécution irrégulière ou dynamique afin que le système d'équilibrage puisse montrer ses capacités de régulation. Une autre possibilité pour créer un déséquilibre serait la création d'une charge parasite extérieure, donc d'une charge d'un processeur qui ne vient pas de l'application test. En effet, en réalité sur un cluster nous trouverons en général plus qu'une application en cours d'exécution. La réaction de l'environnement AMPI sur une charge extérieure devra être aussi testée.
  Ces tests sur l'équilibrage dynamique devraient alors nous permettre d'évaluer les gains et coûts dus aux migrations sur une architecture cluster.
- **Efficacité :** L'environnement AMPI assure des gains de performance par rapport aux programmes traditionnels MPI. Selon les développeurs, ces gains seraient dus au concept de virtualisation des processeurs. Nous allons donc comparer les temps d'exécution d'une version MPI et d'une version AMPI de notre application test pour voir les apports de la virtualisation.

[1] Dans ce chapitre on ne fait plus la différence entre les fonctionnalités d'AMPI, Converse ou encore Charm++. Pour simplifier, nous allons mettre toutes les fonctionnalités applicables sur un programme AMPI comme fonctionnalité de l'environnement AMPI.

[2] Cette application a été développée par l'auteur. (voir aussi `http://users.skynet.be/teller/memoire/`)

– **Conversation :** Un autre intérêt d'écrire ce programme test en deux versions est de voir quelles transformations sont nécessaires pour convertir un programme MPI en AMPI. Remarquons que le manuel d'utilisation d'AMPI propose pour l'écriture d'un programme AMPI de commencer par écrire un programme MPI habituellement et puis de le transformer en AMPI. Cette transformation est aussi analysée dans le chapitre suivant pour une application réelle avec une taille de code beaucoup plus importante que l'application test développée ici.

L'utilité pratique de notre programme test est assez limitée. En effet, il fait uniquement des opérations d'addition (plus 1). Mais l'intérêt d'un tel programme *sum* est d'avoir une simulation d'une charge de calcul, qu'on peut facilement paramétrer pour différentes scénarios de tests.
Par exemple, dans notre programme il doit être facile de modifier la granularité des tâches parallèles, c'est à dire de pouvoir déterminer, tout en gardant une même charge de calcul, si on a beaucoup tâches qui font seulement une petit partie du calcul ou d'avoir quelques grosses tâches qui font une grande partie de la charge de calcul. Aussi, dans les exigences ci-dessus nous avons déterminé que notre programme test devait avoir un déséquilibre. Nous allons voir, que nous avons introduit un paramètre de "charge" qui va créer des irrégularités dans l'application. Par la simplicité de notre programme ces irrégularités sont prévisibles et nous pouvons donc mieux valider la réaction du système d'équilibrage de charge.

Aussi, comme nous allons le voir nous avons choisi un algorithme itératif, avec une synchronisation à la fin de chaque itération. En effet, beaucoup d'application scientifique du calcul parallèle travaille en itération (voir aussi chapitre 5). Souvent la synchronisation à la fin est nécessaire pour échanger des résultats intermédiaires afin de poursuivre le calcul à partir de ces résultats. Comme on a vu, dans AMPI on doit spécifier des moments auxquels un test de migration et des migrations éventuelles peuvent avoir lieu. La fin d'une itération est un moment idéal pour faire appel au test de migration à l'environnement car le nombre de données à migrer avec la tâche est minimal et les données qui doivent migrer avec , donc ceux qui sont encore utilisées dans la prochaine itération, sont facilement identifiables.

Remarquons, qu'on a un peu négligé le facteur de communication, surtout, les données échangées avec les messages ont une taille minimale. En jouant sur la granularité des tâches et le nombre d'itérations, donc de synchronisation, nous pouvons influencer le nombre de communications, mais en général, nos tests ne prennent pas en compte la communication. Vu aussi, que notre architecture test est un simple cluster de PC avec un réseau d'interconnexion Ethernet 100MB/s, nous avons minimisé les communications afin de ne pas trop biaiser nos mesures avec des coûts de communication non négligeables.

4.2 Conception du programme test pour les tests d'ordonnancements

Le problème pratique à résoudre par notre programme peut se spécifier tout simplement : faire un nombre S d'opérations d'additions successives d'entiers 1 et imprimer le résultat de la somme, donc le nombre d'opérations d'additions S, à l'écran.
Pour pouvoir déterminer le niveau de granularité des tâches nous spécifions comme entrée du programme trois paramètres. Le paramètre p indique le nombre de tâches, c'est à dire le nombre de processeurs réels dans la version MPI du programme et le nombre de pro-

cesseurs virtuels dans la version AMPI ; le paramètre n indique le nombre d'itérations du calcul, c'est à dire le nombre de sous-calculs avec synchronisation finale pour communiquer le résultat intermédiaire du sous-calcul et le paramètre m indique le nombre d'opérations d'additions à faire par tâche pendant une itération. Donc le nombre total E d'additions effectuées se calcule par $S = p * n * m$. Par exemple, dans le cas MPI si on a 10 processeurs, et qu'on décide de faire 10 itérations de calcul et que chaque processeur fait 10 000 opérations d'additions lors d'une itération, alors S vaut 1 000 000. Donc la vérification du résultat calculé est triviale.

Remarquons que le standard du *message-passing* permet d'implémenter le modèle SPMD. Nous allons implémenter ce modèle selon un schéma de Maître-Esclave (voir section 1.3.3). Ce schéma a comme avantage d'être simple. Ainsi, nous avons une tâche maître qui reçoit les paramètres initiaux (p, n et m). Comme un processeur réel en MPI ou virtuel en AMPI est réservé au maître, p indique ici respectivement le nombre de processeurs réels ou virtuels dédiés aux esclaves. On doit donc garder, en réalité, $p + 1$ processeurs réels ou virtuels pour le maître et les esclaves.
Ensuite, le maître incite les p esclaves à faire une première itération de calcul de m opérations d'additions. À la fin de cette première itération le maître recueille et additionne les résultats intermédiaires des esclaves et lance une deuxième itération. Le maître répète ce scénario n fois et imprime, à la fin, la somme des opérations effectuées pendant chaque itération.
Avec notre schéma Maître-Esclave on a donc un gestionnaire central qui dirige les itérations et qui recueille les résultats. Dans un modèle classique SPMD l'échange des résultats entre tâches identiques et coopératives n'est pas facile à programmer. En effet, on doit souvent reprendre des mécanismes de cycle ou de shift afin de garantir la transmission des résultats intermédiaires aux bons destinataires. Par exemple, beaucoup d'algorithmes de calcul matriciel utilisent de telles techniques[3]. En plus, le problème de terminaison, c'est à dire savoir quand toutes les tâches ont terminées et que le programme parallèle global peut se terminer, est facile avec un maître. En effet, après n itérations le maître indique aux esclaves de se terminer et peut alors fermer l'environnement parallèle.

Notons, que dans notre calcul simple, les résultats intermédiaires ne sont pas utiles pour la suite des calculs. Ainsi, il n'est pas nécessaire d'implémenter une synchronisation à la fin de chaque itération ; avec des appels non bloquants de communication un esclave pourrait continuer son calcul pendant la transmission du résultat intermédiaire et ainsi combler le temps de communication par des calculs. Mais, ce qui nous intéresse pour nos tests, c'est de voir en premier lieu, les gains de temps avec un équilibrage dynamique et, en deuxième lieu, les effets de la virtualisation, donc de placer plusieurs tâches sur un processeur réel. Pour le premier, si la charge de calcul est mieux équilibrée les temps de calcul d'une itération diminuent. Pour le deuxième, les attentes de communication sont comblées par le calcul d'une autre tâche.

Afin de bien comprendre l'exécution de notre programme, développons maintenant le pseudo-code pour la partie maître et esclave (la partie initialisation de l'environnement parallèle et le saisi des paramètres a été négligé) :

- **Maître**
  p = nombre d'esclaves
  n = nombre d'itérations
  s_j = résultat intermédiaire de l'esclave j (pour $j \in \{1, \ldots, p\}$)
  S = somme totale des résultats intermédiaires

[3] Voir aussi [30].

$S = 0$
prendre *temps-avant*
for $i = 0$ to $n - 1$
    envoyer à tout le monde : *calculer*
    recevoir de tout le monde : s_j (pour $j \in \{1, \ldots, p\}$)
    $S = S + \sum_{j=0}^{p} s_j$
prendre *temps-après*
calculer *temps* = *temps-apres* - *temps-avant*
afficher S et *temps*
envoyer à tous le monde : *terminer*

- **Esclave**
  m = nombre d'opérations d'additions à effectuer par itération
  s = résultat intermédiaire

  recevoir *commande* de maître
  while (commande == calculer) //sinon commande = terminer
      $s = 0$
      for $i = 0$ to $m - 1$
          $s = s + 1$
      envoyer s à maître
      recevoir *commande* de maître

Le pseudo-code donné ici spécifie ici une application très régulière. En effet, la charge de chaque esclave est identique et ne change pas d'une itération à l'autre. Afin de créer, comme exigé, une application irrégulière, nous allons introduire un concept de *charge* dans le code de l'esclave.
Notre *charge* est simulé par un entier. Cet entier va multiplier le nombre d'opérations d'addition effectué par l'esclave pendant une itération (voir pseudo-code en-dessous). Par exemple, si *charge* vaut 2 notre esclave répète deux fois les m opérations d'addition. Ainsi on a doublé sa charge.

- **Esclave**
  ...
  while (commande == calculer) //sinon commande = terminer
      for $k = 1$ **to** ***charge***
          $s = 0$
          for $i = 0$ to $m - 1$
              $s = s + 1$
      envoyer s à maître
      recevoir *commande* de maître

Si nous fixons la charge à 1 pour toutes les esclaves et tous les itérations, nous avons notre application régulière initiale. Avec cette application régulière, nous pouvons par exemple tester l'influence d'une charge extérieure sur l'exécution de l'application dans l'environnement MPI et AMPI. Afin de créer des charges différentes sur chaque esclave, nous pouvons fixer la charge d'un esclave équivalent à son rang : c'est à dire dans le cas ou p est 10 l'esclave numéro 1 à une charge de 1 mais l'esclave numéro 10 à une charge de 10 est a donc 10 fois plus de charge de calcul que l'esclave numéro 1. Ainsi, nous avons un code identique qui crée une charge différente dépendante de l'exécution.
Aussi, nous pouvons ici créer des charges qui évoluent dynamiquement pendant l'exécution. Nous pouvons simuler ceci en affectant une charge selon le rang des esclaves et en inversant par exemple cette charge à la moitié des itérations. Par exemple, si nous avons 10 esclaves, alors l'esclave numéro 1 à une charge de 1 pendant la première moitié des itérations et une charge de 10 pendant la deuxième moitié des itérations :

– **Esclave**

```
...
iter = 0
while (commande == calculer)
    if (iter < (n/2))    //première moitié des itérations
        charge = monRang ;
    else    //deuxième moitié des itérations
        charge = (nbrEscalve + 1) - monRang ;
    for k = 1 to charge
    ...
    iter = iter + 1
```

Pour la gestion de la migration en AMPI, nous allons de nouveau utiliser le maître comme coordinateur. Entre deux itérations c'est lui qui va décider d'un appel de migration. Pour cela, il envoie alors aux esclaves l'indication pour un appel de migration, afin que toutes les tâches fassent un appel système bloquant au même moment. Ainsi, notre implémentation est générique, et nous pouvons gérer les tests de migration avec un paramètre *LBPeriod* qui indique après combien d'itérations, un test de migration est effectué. Par exemple, si *LBPeriod* vaut 2, notre programme fait des tests de migration chaque deuxième itération. Nous pouvons ainsi diminuer les coûts dus aux tests de migration, mais un déséquilibre éventuel sera traité plus lentement car des migrations ne peuvent avoir lieu qu'à chaque *LBPeriod*.

Selon notre exigence de conversion, regardons plus précisément les changements nécessaires pour transformer un programme MPI en AMPI. Faisons donc un parcours des transformations proposées dans le manuel AMPI [1]).
Les programmes MPI traditionnels considèrent qu'à chaque processeur est uniquement accordé un processus MPI. Ainsi, ils considèrent qu'il n'existe qu'un flux de contrôle par processeurs. En AMPI, il existe plusieurs *user-level threads* (les processeurs virtuels) par processeur réel. Il peut donc avoir interférence entre ces différents processeurs virtuels. Une transformation majeure est de rendre privées les variables globales, pour les protéger d'accès concurrents.
Notons que dans notre programme *sum* les variables globales, ne sont pas de vraies variables. Elles stockent uniquement les paramètres de l'appel du programme et ne sont plus modifiées par après. Le risque d'une erreur, en les traitant en parallèle, est donc éliminé. Pour le test, veillons quand même à les protéger d'accès concurrents. Il suffit de placer toutes les variables globales dans une structure, puis, de créer une variable avec ce type de structure dans la fonction *main* et de passer cette variable dans les sous-routines qui utilisent une variable globale. Il va de soi, que l'appel des différentes variables globales doit être remplacé par un appel d'une variable en structure de données. Pour notre programme *sum* la structure s'énonce comme suite :

```
struct shareddata{
  int nbrProc;      /* nombre de processus */
  double m;         /* nombre d'opérations par itération par esclave */
  double n;         /* nombre d'itérations */
  int periodLB;     /* nombre d'itérations entre deux tests de migration*/
};
```

Outre ce changement, l'environnement AMPI ne peut décider efficacement du bon moment pour effectuer une migration, l'étape d'initialisation n'est pas judicieux, car le comportement du programme à ce stade n'est pas représentatif pour la suite. Le programmeur connaissant le mieux son programme peut via l'appel MPI_Migrate() initier un test de migration. Comme nous l'avons vu, un bon moment pour faire appel à un tel test est

la fin d'une itération.
Il reste à migrer l'état actuel (valeurs des variables) du *thread*. Pour les variables locales (*stack data*) l'AMPI propose un transfert automatique via l'utilisation du même espace adressage virtuel (*iso-adresse*). Aussi pour les variables allouées dynamiquement (*heap data*) un procédé d'*iso-adresse* est proposé [11]. Or, pour les variables du *heap* cette technique limite fortement l'espace total disponible pour tous les processeurs virtuels. En plus, elle est encore en développement et a montré des instabilités pour les variables allouées dynamiquement avec le programme *sum*.
L'alternative manuelle est de communiquer à l'environnement AMPI la grandeur des variables, comment les compresser ("pack" : sérialisation et libération de la mémoire) pour envoyer via un message et comment les décompresser ("unpack" : allocation de la mémoire et désérialisation) pour la destination. Ceci est réalisé via une implémentation d'une fonction *pup* et de son enregistrement auprès de l'environnement avec la variable allouée dynamiquement qui est sensée être migrable.
Voici, un exemple d'une telle fonction *pup* qui a un comportement différent (dirigé avec deux tests) selon s'il s'agit d'une compression ou d'une décompression :

```
void shareddataPup(pup_er p, struct shareddata **cpc){
  struct shareddata *c;

  //si décompression
  if (pup_isUnpacking(p)){
    //alors allocation de mémoire chez le nouveau processeur
    *cpc = (struct shareddata *)malloc(sizeof(struct shareddata));
  }

  //sérialisation et désérialisation
  c = *cpc;
  pup_int(p,&c->nbrProc);
  pup_double(p,&c->m);
  pup_double(p,&c->n);
  pup_int(p,&c->periodLB);

  //si compression
  if(pup_isPacking(p)){
    // alors libération de mémoire chez l'ancien processeur
    free(c);
  }
}
```

Ainsi, la fonction de registration s'énonce :

```
struct shareddata *c;
c = (struct shareddata *)malloc(sizeof(struct shareddata));
MPI_Register(&c,(MPI_PupFn) shareddataPup);
```

Le code MPI et AMPI de notre application se trouve à l'annexe D.

4.3 Les séries de test

Afin, d'évaluer l'équilibrage de charge et le principe de virtualisation nous avons comparé une version MPI de notre programme avec une version AMPI. La version AMPI a été testée avec quatre stratégies de migration fournies par les développeurs de l'environnement AMPI :

- **Stratégie vide :** Cette stratégie n'est pas une stratégie réelle et ne fournit pas une gestion de migration. Ainsi, elle absorbe chaque demande de test de migration et l'exécution du programme AMPI se fait alors sans migration.
- **RefineLB :** Une stratégie centrale défensive, qui migre des tâches d'un processeur sur-chargé vers des processeurs moins chargés jusqu'à ce que la charge de tous les processeur soient en-dessous d'un seuil critique. Cette stratégie ne prend pas en compte la charge extérieure.
- **GreedyLB :** C'est une stratégie centrale agressive, qui fait une re-allocation de toutes les tâches afin d'équilibrer au mieux la charge. Cette stratégie ne prend pas en compte la charge extérieure.
- **WSLB :** Cette stratégie distribuée est faite pour un réseau de stations de travail (*WorkStation*). Elle équilibre la charge en prenant compte de la charge extérieure.

L'architecture de test était le pool des étudiants de spécialité "Ingénieur Informatique et Gestion" à la FPMS. Ce pool est constitué d'un réseau de PC Linux avec des processeurs Intel Pentium 4 de 2.00 Ghz et 256 Mo de Ram. Les ordinateurs sont connectés via un Ethernet 100 Mb/s. Comme implémentation du standard MPI, nous avons utilisé l'environnement MPICH.

En pratique nous avons fait quatre séries de tests qui jouent sur le paramètre *charge* de notre application *sum* :

- **Charge Void :** Une première série de tests évalue les deux versions de *sum* avec les quatre stratégies de migration pour la version avec le paramètre de *charge* égal à 1 et sans charges extérieures ; donc nous avons une application totalement régulière.
- **Charge Rang :** Les tests de cette deuxième série évalue les performances des environnements parallèles pour une charge déterminée pendant l'exécution ; la charge est fixée proportionnellement au rang des esclaves.
- **Charge Dynamique :** La troisième série implémente l'idée d'une charge dynamique donc évolutive pendant l'exécution. La charge est aussi ici basée sur le rang mais inversée pour la deuxième partie des itérations.
- **Charge Extérieure :** Comme dans la première série la charge de cette série est constante pour tous les esclaves, mais on va simuler une application extérieure qui utilise 66% de CPU d'un seul processeur. Les autres processeurs n'ont pas cette charge extérieure.

Pour chaque série nous avons exécuté nos versions pour calculer 10 milliards et 100 milliards d'opérations d'additions sur un cluster de trois ordinateurs du pool. Les paramètres pour les tests étaient :

version MPI (2 Esclaves)				
S : nbr op. d'add.	**p : nbr escl.**	**n : nbr itér.**	**m : nbr op./itér.**	
10 000 000 000	2	10	500 000 000	
100 000 000 000	2	10	5 000 000 000	

version MPI (10 Esclaves)				
S : nbr op. d'add.	**p : nbr escl.**	**n : nbr itér.**	**m : nbr op./itér.**	
10 000 000 000	10	10	100 000 000	
100 000 000 000	10	10	1 000 000 000	

version AMPI				
S : nbr op. d'add.	**p : nbr escl.**	**n : nbr itér.**	**m : nbr op./itér.**	**LBperiod**
10 000 000 000	10	10	100 000 000	1
100 000 000 000	10	10	1 000 000 000	1

Notons qu'on a fait des tests avec la version MPI avec 2 et 10 esclaves. En effet, dans des programmes traditionnels de MPI on affecte un processus MPI par processeur. Comme nos tests se font avec trois processeurs nous en avons un pour le maître et deux pour les esclaves. Mais, pour avoir deux exécutions équivalentes dans les tests où la *charge* dépend du rang des esclaves nous avons aussi travaillé avec 10 esclaves en MPI. Pour cela, nous avons utilisé une fonctionnalité de l'implémentation MPICH. Semblable au principe de virtualisation, MPICH permet de placer plusieurs processus MPI sur un processeur. Ce sont alors des processus gérés par le système d'exploitation, c'est à dire que si on a trois esclaves sur un processeur, alors on trouve dans le gestionnaire de tâche du système d'exploitation trois processus MPI. Au contraire pour les *user-level threads* d'AMPI, le système d'exploitation ne connaît qu'un seul processus AMPI, même s'il existe trois esclaves sur le processeur.
En plaçant pour MPI, comme en AMPI, 10 esclaves sur trois processeurs, on peut avoir une situation initiale de distribution de charge identique et ainsi, nous pouvons mieux mesurer les gains dus à l'équilibrage de charge d'AMPI par rapport à MPI.

4.4 Résultats et interprétations

Dans cette section nous allons analyser les résultats des différentes séries de test. Cette analyse explique quelques interprétations et conclusions.
Ces interprétations ne sont pas toujours faciles. En effet, comme les différentes mesures de temps vont le montrer, il y a parfois de fortes différences entre le temps d'exécution d'un programme dans le même contexte. Néanmoins, si nous regardons les temps des itérations ou les fichiers *log* que l'environnement AMPI crée, nous pouvons tirer des conclusions intéressantes.
Les résultats et interprétations présentés ici sont alors complétés par les tests sur une application réelle dans le chapitre suivant.

4.4.1 Charge Void

Les temps d'exécution de notre application régulière ($charge = 1$) dans un contexte équilibré (sans charge extérieure) nous montrent des indications sur la performance du principe de virtualisation. Les meilleurs temps sont mis en gras :

Charge Void (temps d'exécution du programme en seconde)							
nbr. op.*	C **	MPI 2 escl.	MPI 10 escl.	AMPI Str. vide	AMPI RefineLB	AMPI WSLB	AMPI GreedyLB
10	137	68	**54**	**56**	69	69	
10	/	68	**54**	**57**	60	63	69
10	/	68	**54**	97	68	68	/
100	1 368	679	**537**	**565**	**562**	584	686

(* Nombre d'opérations en milliards.)
(** Exécution séquentielle sur un processeur.)

D'abord comparons les temps d'exécution de la version MPI de deux esclaves avec la version AMPI. La version AMPI est, en général, plus rapide ; AMPI profite ici du principe de virtualisation. AMPI peut placer 10 esclaves sur les trois processeurs. Il place aussi des esclaves sur le processeur où se trouve le maître. Dans notre architecture Maître-Esclave, le maître attend pendant que les esclaves font leurs calculs, donc il y a un appel

bloquant. Comme chez AMPI il y a au moins un esclave sur le processeur du maître, ce temps d'attente du maître est rempli avec le calcul de l'esclave. Ainsi, le temps global est plus petit en AMPI qu'en MPI2 esclaves où le calcul des esclaves est uniquement fait par les deux processeurs qui n'hébergent pas le maître.
Le fait de placer pour MPI le travail de calcul des esclaves sur deux processeurs et pour AMPI sur trois processeurs à cause de la virtualisation donne naturellement un avantage pour AMPI. Afin, de valider les autres caractéristiques d'AMPI nous avons éliminé cet avantage de la distribution de travail à cause de la virtualisation. Une solution non réalisée dans le cadre de ces tests était d'augmenter le nombre de processeurs réels participant à l'expérience. Ainsi, la distribution de la charge initiale de tous les processeurs entre les versions MPI et AMPI soient plus proches et les différentes temps d'exécutions sont comparables réellement. Pour la suite des tests, nous avons utilisé une autre solution qui place aussi pour MPI 10 esclaves sur les trois processeurs. Nous avons de suite vérifié que la distribution initiale des charges était identique pour un programme MPI ou AMPI. Ainsi, l'avantage à cause de la virtualisation est éliminé.
Pour notre charge void AMPI est plus lent avec la version MPI avec 10 esclaves sur trois processeurs. Cela est dû au sur-coût d'emballage d'un processus MPI dans l'environnement AMPI. Remarquons encore que l'utilisation d'une stratégie de migration engendre encore un sur-coût supplémentaire dû à la gestion de la stratégie.
En bref, pour une application régulière, dans un contexte équilibré et avec une implémentation de MPI qui a aussi des capacités de virtualisation l'utilisation d'AMPI n'est pas pertinente.

Pour le troisième test avec 10 milliard d'opérations et dans certains tests de ci-dessous, l'environnement AMPI avec la stratégie vide montre un temps d'exécution anormalement long. Une des causes pourrait être des facteurs externes, comme la charge du réseau. (Le réseau utilisé n'était pas un réseau à l'utilisation exclusive pour l'application test.) Le facteur du trafic sur le réseau qui peut influencer les mesures ne pouvait pas être contrôlé complètement. Certains résultats sont difficilement reproductibles et des mesures significativement différentes pour deux test identiques sont à interpréter avec précaution et le plus d'objectivité possible. Le comportement des stratégies d'AMPI n'est pas déterministe d'une exécution à l'autre et, les migrations ne sont pas totalement identiques d'une exécution à une autre ce qui provoque des temps d'exécution différents.

4.4.2 Charge Rang

Charge Rang (temps d'exécution du programme en seconde)					
nbr. op.*	MPI 10 escl.	AMPI Str. vide	AMPI RefineLB	AMPI WSLB	AMPI GreedyLB
10	366	481	353	**287**	
10	369	418	**320**	340	352
10	367	383	353	**294**	/
100	3 661	3 758	3 583	**3 311**	3 650

(* Nombre d'opérations en milliards.)

En partant d'une situation initiale identique pour AMPIet MPI(10 esclaves) l'utilité d'AMPI avec des stratégies de migration qui prennent en compte la charge de chaque tâche pour une application irrégulière est bien démontrée dans les temps d'exécution du tableau. Par exemple, regardons ce qui se passe pour la stratégie WS dans la figure 4.1. Sur chaque ligne de temps d'un processeur (PE), on représente les temps d'exécution d'un

esclaves avec des rectangles rouges. Les différentes longueurs des rectangles montrent les différentes charges de calcul de chaque esclave. A l'itération 0, nous voyons la distribution initiale des esclaves. Cette première itération montre aussi de longs rectangles hachurés en blanc et noir. Ces zones correspondent à des temps d'inoccupation du processeur. A l'itération 1 ces zones sont beaucoup plus petites ; à la fin de l'itération 0 il y a eu deux migrations : l'esclave 5 a migré de PE 1 à PE 0 et l'esclave 9 a migré de PE 2 à PE 0. A l'itération 2 la charge est encore mieux équilibrée, car encore moins de zone hachurées ; l'esclave 1 a encore migré de PE 0 à PE 1. Avec l'équilibration des charges nous remarquons aussi que les temps d'exécution d'une itération diminuent. L'itération 1 est donc beaucoup plus rapide que l'itération 0.

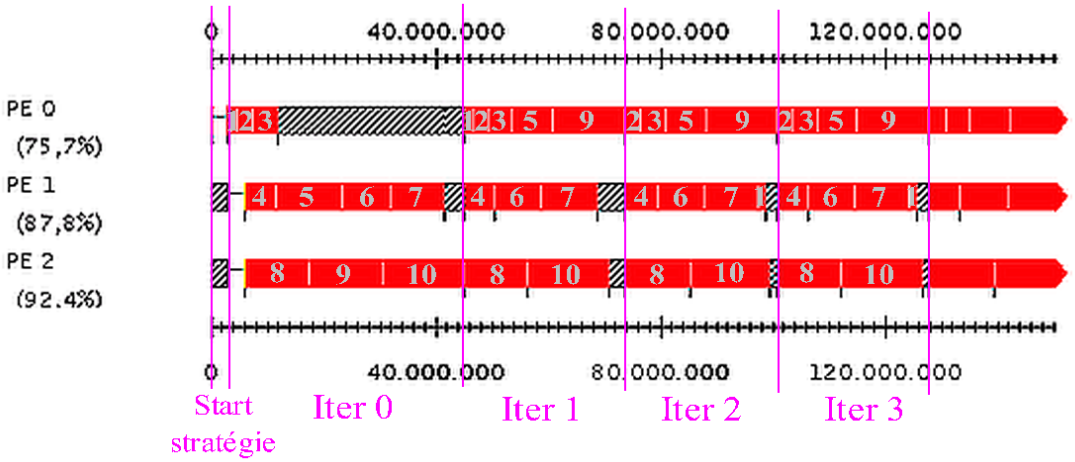


FIG. 4.1 – *Lignes du temps des trois processeurs pour la version* AMPI *avec la stratégie WS du programme sum avec la charge équivalente au rang de l'esclave.*(figure faite avec l'outil Projection de CHARM++ à partir des fichiers log d'AMPI)

Pour nos temps d'exécution les stratégies montrent parfois des performances différentes dans le même contexte d'exécution (voir les trois premiers tests avec 10 milliard d'opérations). Ces différence sont dues à l'exécution non déterministe des stratégies. En effet, les migrations effectuées par les stratégies ne sont pas toujours identiques d'une exécution à l'autre. Cela résulte finalement dans des charges plus ou moins bien équilibrées qui provoquent des irégulations dans les temps totaux d'exécution. Néanmoins, la stratégie *greedy* migre à cause de sa nature agressive un grand nombre de tâche, ce qui provoque des sur-coûts à cause des migrations et des petits déséquilibres plus importants que dans les deux autres stratégies. Ainsi, la stratégie *Greedy* est la moins performante des trois stratégies réelles.

Remarquons que des absences[4] de temps pour la stratégie *Greedy* sont dues à une instabilité de l'environnement AMPI. L'environnement AMPI est encore en stade de développement. Parfois pendant une phase de migrations, l'application test se terminait avec une erreur.

[4]Dans le dernier test avec 10 milliards opérations la stratégie *Greedy* n'était plus testée.

4.4.3 Charge Dynamique

Charge Dynamique (temps d'exécution du programme en seconde)					
nbr. op.*	MPI 10 escl.	AMPI Str. vide	AMPI RefineLB	AMPI WSLB	AMPI GreedyLB
10	408	597	**305**	325	511
10	406	684	**333**	382	
10	409	441	**303**	344	/
100	4 062	3 865	3 445	**3 233**	3 377

(* Nombre d'opérations en milliards.)

Au niveau des temps d'exécution avec une charge dynamique nous avons les mêmes résultats que dans la série de tests précédents. Examinons plus en détail les différents temps d'itération de notre programme *sum* (figure 4.2). D'abord notons que, même avec une charge qui évolue dynamiquement, les temps d'itération pour MPI et AMPI sont ici relativement constants. Cela est dû à un hasard de l'application et des distributions des esclaves sur les processeurs. En effet, l'inversement de la charge des esclaves revient dans ces deux cas à un échange de deux groupes esclaves entre deux processeurs (1, 2, 3 de PE 0 à PE 2 et 8, 9, 10 de PE0 à PE 1). Le processus PE 0, le moins chargé avant l'inversement de charge, reçoit donc la charge que PE 2 avait et inversement. Le processeur le plus chargé qui détermine le temps d'exécution d'une itération est un autre après l'inversement mais sa charge est la même que celle du processeur déterminant le temps auparavant et comme les puissances de nos trois processeurs sont identiques le temps d'une itération est inchangé. À cause de différents tests pour simuler la charge dynamique, les temps d'exécution de MPI sont plus élevés que dans la série de tests précédente.

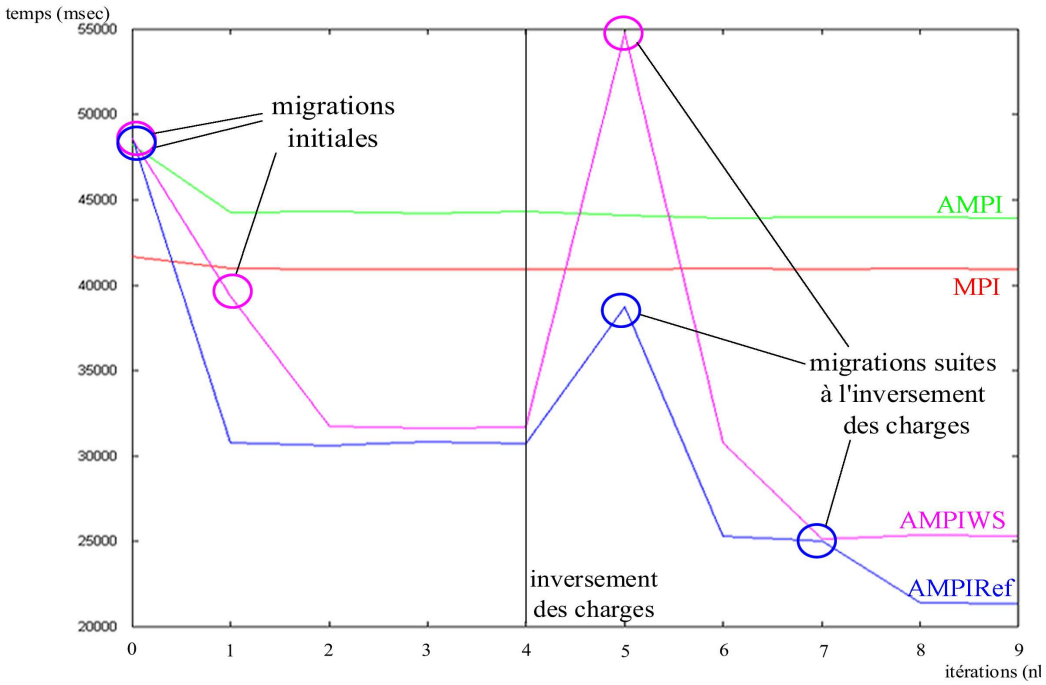


FIG. 4.2 – *Courbes des temps d'itération des différentes versions pour le programme sum avec charge dynamique*

Les versions AMPI avec les deux stratégies montrent des courbes très intéressantes. Ainsi, à la fin de l'itération 0 et de l'itération 1 ces deux stratégies procèdent à des migrations qui vont descendre les temps d'itération bien en-dessous des temps d'itération de MPI. Après l'itération 4 l'inversement des charges à lieu et provoque un grand déséquilibre pour l'itération 5 un temps d'itération élevé. Les deux stratégies réagissent promptement en rétablissant l'équilibre avec quelques migrations, ce qui a pour conséquence une chute des temps d'itération.
Remarquons aussi que les temps d'itération pour les versions avec ces stratégies sont plus bas après la deuxième équilibration. Cela s'explique par le fait, que les stratégies ont trouvé, à partir de la distribution du déséquilibre, un équilibre plus parfait que pendant la première vague de migrations. Nous voyons aussi que la stratégie WS réagit toujours un peu plus lentement (à cause de sa nature distribuée) que la stratégie *Refine* qui elle, est centrale et a une vue globale.

4.4.4 Charge Extérieure

Pour cette série de tests avec une charge extérieure sans surprise, la seule stratégie qui prenne en compte donne la meilleure performance :

Charge Extérieure (temps d'exécution du programme en seconde)					
nbr. op.*	MPI-2 2 escl.	AMPI Str. vide	AMPI RefineLB	AMPI WSLB	AMPI GreedyLB
10	207	129	130	**104**	272
10	207	192	174	**102**	282
10	207	128	170	**109**	/
100	2 086	1 259	1 303	**1 093**	1 567

(* Nombre d'opérations en milliards.)

Le graphique 4.3 montre bien l'influence de la charge extérieure sur les temps d'exécution des esclaves de PE 2. La migration de l'esclave 8 du processeur PE 2 au processeur PE 0 à la fin de l'itération 0 diminue le temps d'exécution des itérations et équilibre mieux les charges. Vu que le temps d'exécution d'un esclave diminuait fortement sur un processeur sans charge extérieure, la stratégie pouvait encore faire mieux en migrant encore un esclave du processeur PE 2 . Mais, comme les estimations des temps d'exécution se basent sur le principe de persistance, la stratégie WS estime qu'une migration aggravent la situation. En effet, si le temps d'exécution d'un esclave du processeur PE 2 ne diminuait pas en s'exécutant sur un autre processeur, le temps global d'une itération augmentait car les boites hachurées du temps d'inoccupation des processeurs PE 0 et PE 1 sont plus petits que la boite de temps d'exécution de l'esclave 9 ou 10.

Finalement, pour profiter par exemple d'une optimisation comme demander dans le dernier paragraphe AMPI permet l'écriture d'une stratégie de migration par le programmeur de l'application. Ainsi, la stratégie peut respecter la nature propre d'une application ou de son contexte d'exécution.

4.5 Premières conclusions

Les tests effectués ici ont montré l'intérêt d'AMPI pour une application dynamique dans un contexte hétérogène par exemple un cluster avec charge extérieure.
La conversion d'un code MPI en AMPI était d'un effort limité pour cette application test.

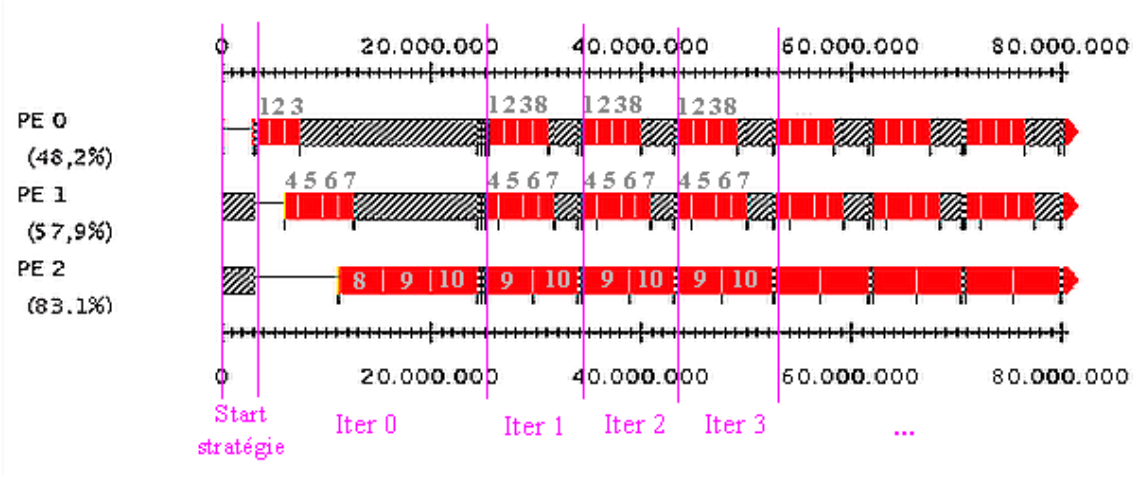


FIG. 4.3 – *Lignes de temps des trois processeurs pour la version* AMPI *avec la stratégie WS du programme sum avec la charge extérieure.*(figure faite avec l'outil Projection de CHARM++ à partir des fichiers log d'AMPI)

Mais, la première série de tests sans déséquilibre a montré que le concept de virtualisation n'était pas limité à un environnement AMPI ou CHARM++.
Dans le chapitre suivant nous allons nous poser les mêmes questions, mais dans un cadre plus réel.

Chapitre 5

L'application pratique
-
Le programme *press*

Ce dernier chapitre va d'une part montrer les avantages d'un équilibrage dynamique de charge dans le cadre d'une application réelle et d'autre part, les difficultés d'un passage d'un code MPI plus complexe que le code de notre simple programme test en AMPI sont investiguées.

La dynamique moléculaire fournit des tas de problèmes qui se traitent bien pour l'équilibrage dynamique de charge. Nous avons eu la chance de pouvoir traiter, dans le cadre du stage, une simulation de ce domaine.
Pour comprendre le déroulement et les résultats du programme *press* qui fait l'objet de cette simulation de physique, une première section explique les grands concepts de la dynamique moléculaire. Dans le même but, une deuxième partie décrit la simulation en elle-même.
Comme le code de la simulation était écrit en séquentiel nous allons approfondir dans une section la conception pour paralléliser le programme *press*. Pour passer finalement à une description du programme *press* parallélisé.
La section qui est la plus importante est la dernière. Elle traite le double objectif de ce chapitre, en analysant et interprétant différents résultats intéressants des tests qui ont été réalisés avec le programme *press*.

5.1 La dynamique moléculaire

Expliquons le domaine de la dynamique moléculaire qui forme le cadre de notre application pratique.
La dynamique moléculaire [47] peut être définie comme une des techniques de simulation numérique exploitées pour étudier les propriétés (température, pression, ...) des systèmes de particules [1]. C'est une étude des propriétés macroscopiques de la matière à partir des caractéristiques microscopiques de ses atomes et molécules.

Le principe des simulations numériques consiste à reproduire les configurations d'un ensemble de particules grâce à l'ordinateur. En dynamique moléculaire, les équations du mouvement de la mécanique classique sont intégrées numériquement et le déplacement de

[1] En dynamique moléculaire les systèmes de particules sont des ensembles d'atomes ou de molécules.

chaque particule est enregistré ; on en déduit ensuite des propriétés macroscopiques.

L'étude analytique des systèmes de particules en interaction est une tâche très complexe. Même avec l'introduction de certaines approximations leur résolution consomme beaucoup de ressources de calcul. Le temps de calcul joue surtout un rôle primordial. Ainsi, la simulation numérique impose l'utilisation des techniques de calcul parallèle.

5.1.1 Les principes de la dynamique moléculaire[2]

Équations de Newton

En premier lieu, la dynamique moléculaire résout les équations classiques du mouvement pour un ensemble de N particules interagissant :

$$\vec{f_i}(t) \quad = \quad m_i \vec{a_i}(t) \tag{5.1}$$

où t représente le temps, $\vec{f_i}$ la force appliquée sur la particule i, m_i sa masse et $\vec{a_i}$ son accélération.

Remarquons que le système d'équations (5.1) est déterministe. Dès que les positions et vitesses initiales des N particules sont connues, les positions et vitesses sont connues à tout moment ultérieur.

L'énergie potentielle

Le potentiel U a une importance particulière en dynamique moléculaire. En effet, la force $\vec{f_i}$ (5.1) peut être obtenue en dérivant le potentiel U par rapport à la position $\vec{r_i}$ de la particule i :

$$-\frac{dU}{d\vec{r_i}} \quad = \quad \vec{f_i}(t) \tag{5.2}$$

En calculant l'énergie potentielle, on peut donc déterminer la force qu'une particule du système subit. La dynamique moléculaire utilise un ajustement empirique pour le calcul de l'énergie potentielle. Une première partie de l'ajustement est le potentiel d'interaction entre atomes qui est constitué principalement du potentiel de Lennard-Jones[3]. Un deuxième terme de l'ajustement est le potentiel intramoléculaire ou harmonique qui représente la contribution supplémentaire des liaisons chimiques entre les atomes d'une molécule à l'énergie potentielle.

5.1.2 La résolution numérique en dynamique moléculaire

La dynamique moléculaire utilise une résolution numérique qui exploite la méthode des différences finies. Le principe consiste à déterminer les positions, vitesses et accélérations à un instant $(t + \Delta t)$ à partir de leurs valeurs à l'instant (t).[4] C'est donc une méthode itérative.

En général l'organigramme donné à la figure 5.1 synthétise les différentes étapes d'une résolution numérique en dynamique moléculaire [21]. Dans les sections suivantes, nous allons détailler ces différentes étapes.

[2]Pour plus d'informations sur la théorie physique utilisée voir Annexe B.

[3]Expression qui rend compte le potentiel d'interaction entre un paire d'atomes et qui dépend uniquement de la nature des atomes interagissant et de la distance séparant les deux atomes.

[4]Un pas de temps Δt typique est de l'ordre de 10^{-15} s.

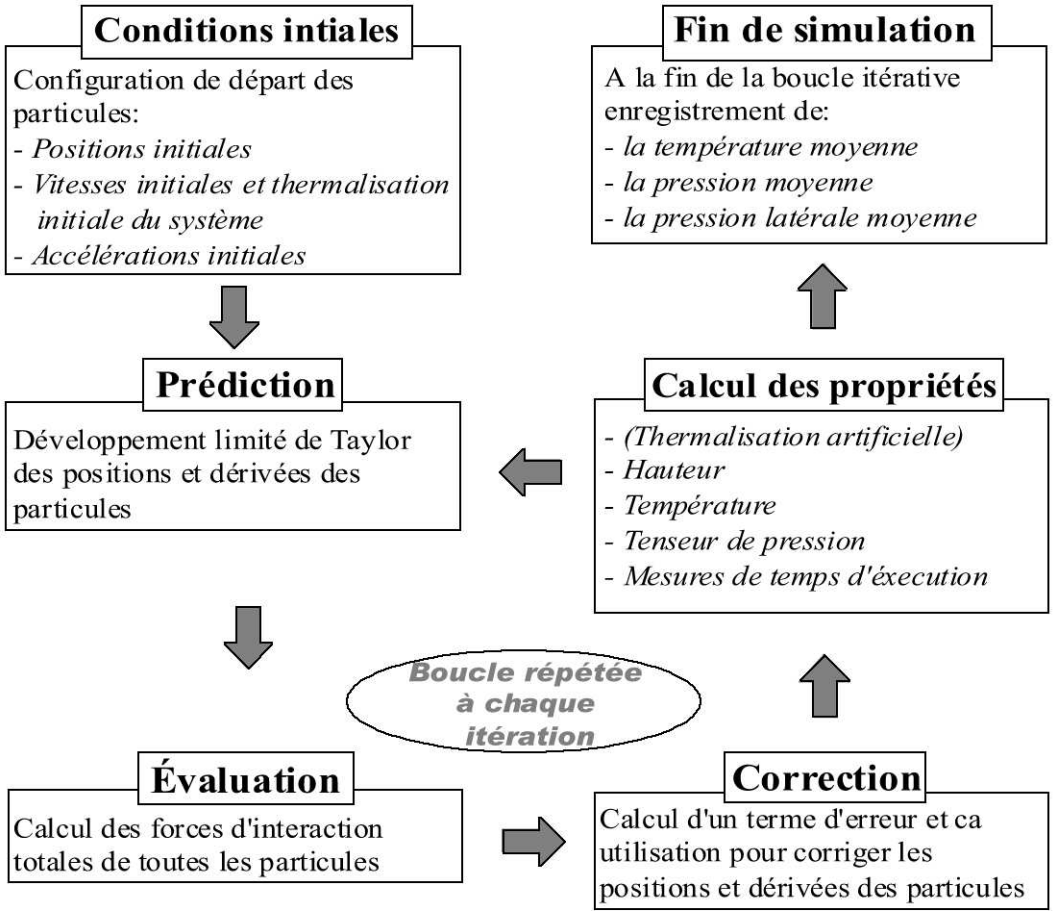


FIG. 5.1 – *Organigramme du code de dynamique moléculaire développé*
(Les termes en italique sont spécifiques à l'application développée dans ce chapitre)

Prédiction-Correction

Le coeur du programme représenté à la figure 5.1 est symbolisé par une boucle itérative dont chaque itération représente un pas de temps discret.
L'algorithme de la dynamique moléculaire consiste à résoudre les équations de Newton (5.1) en accord avec l'expression du potentiel préalablement choisi (équation 5.2). En général, un algorithme de "prédiction-correction"[19] de cinquième ordre est utilisé.

L'algorithme s'articule autour de trois étapes[5] :

1. **La prédiction**
   Les positions, les vitesses et les accélérations par rapport au temps à l'instant $(t+\Delta t)$ sont estimées par un développement en série de Taylor à partir de leurs valeurs à l'instant t.

$$\vec{r}_i^{\,p}(t+\Delta t) = \vec{r}_i(t) + \vec{v}_i(t)\Delta t + \vec{a}_i(t)\frac{(\Delta t)^2}{2!} + \vec{a}_i^{(3)}(t)\frac{(\Delta t)^3}{3!} + \vec{a}_i^{(4)}\frac{(\Delta t)^4}{4!} + \vec{a}_i^{(5)}(t)\frac{(\Delta t)^5}{5!}$$

[5]Les équations mathématiques sont données ici à titre illustratif de l'algorithme de prédiction-correction. Le lecteur non-familiarisé avec le domaine mathématique peut retenir que les séries de Taylor permettent le calcul d'une approximation de la solution recherchée.

$$
\begin{aligned}
\vec{v}_i^{\,p}(t+\Delta t) &= \vec{v}_i(t) + \vec{a}_i(t)\Delta t + \vec{a}_i^{(3)}(t)\frac{(\Delta t)^2}{2!} + \vec{a}_i^{(4)}(t)\frac{(\Delta t)^3}{3!} + \vec{a}_i^{(5)}\frac{(\Delta t)^4}{4!} \\
\vec{a}_i^{\,p}(t+\Delta t) &= \vec{a}_i(t) + \vec{a}_i^{(3)}(t)\Delta t + \vec{a}_i^{(4)}(t)\frac{(\Delta t)^2}{2!} + \vec{a}_i^{(5)}(t)\frac{(\Delta t)^3}{3!} \\
\vec{a}_i^{\,p(3)}(t+\Delta t) &= \vec{a}_i^{(3)}(t) + \vec{a}_i^{(4)}(t)\Delta t + \vec{a}_i^{(5)}(t)\frac{(\Delta t)^2}{2!} \\
\vec{a}_i^{\,p(4)}(t+\Delta t) &= \vec{a}_i^{(4)}(t) + \vec{a}_i^{(5)}(t)\Delta t \\
\vec{a}_i^{\,p(5)}(t+\Delta t) &= \vec{a}_i^{(5)}(t)
\end{aligned}
$$

2. **L'évaluation**
   La force $\vec{f}_i$ (équation 5.2) est calculée à l'instant $(t+\Delta t)$ à partir de l'expression du potentiel U et des positions prédites à la première étape.

$$
\vec{f}_i(t+\Delta t) = -\frac{dU(t+\Delta t)}{d\,\vec{r}_i^{\,p}(t+\Delta t)}
$$

3. **La correction**
   Un terme d'erreur est calculé à partir de deux accélérations, l'une estimée par la prédiction, l'autre (équation 5.1) calculée à partir de la force $\vec{f}_i$ issue de l'étape d'évaluation. Cette "erreur" est utilisée pour corriger les positions, les vitesses et les accélération prédites.

   Les deux accélérations :

$$
\begin{aligned}
&\vec{a}_i^{\,p}(t+\Delta t) \\
&\vec{a}_i(t+\Delta t) = \frac{\vec{f}_i(t+\Delta t)}{m_i}
\end{aligned}
$$

   Le terme d'erreur vaut :

$$
\Delta\vec{a}_i = \vec{a}_i(t+\Delta t) - \vec{a}_i^{\,p}(t+\Delta t)
$$

   Les équations de correction s'écrivent :

$$
\begin{aligned}
\vec{r}_i(t+\Delta t) &= \vec{r}_i^{\,p}(t+\Delta t) + \alpha_0\Delta\vec{a}_i \\
\vec{v}_i(t+\Delta t)\Delta t &= \vec{v}_i^{\,p}(t+\Delta t)\Delta t + \alpha_1\Delta\vec{a}_i \\
\vec{a}_i(t+\Delta t)\frac{(\Delta t)^2}{2!} &= \vec{a}_i^{\,p}(t+\Delta t)\frac{(\Delta t)^2}{2!} + \alpha_2\Delta\vec{a}_i \\
\vec{a}_i^{(3)}(t+\Delta t)\frac{(\Delta t)^3}{3!} &= \vec{a}_i^{\,p(3)}(t+\Delta t)\frac{(\Delta t)^3}{3!} + \alpha_3\Delta\vec{a}_i \\
\vec{a}_i^{(4)}(t+\Delta t)\frac{(\Delta t)^4}{4!} &= \vec{a}_i^{\,p(4)}(t+\Delta t)\frac{(\Delta t)^4}{4!} + \alpha_4\Delta\vec{a}_i \\
\vec{a}_i^{(5)}(t+\Delta t)\frac{(\Delta t)^5}{5!} &= \vec{a}_i^{\,p(5)}(t+\Delta t)\frac{(\Delta t)^5}{5!} + \alpha_5\Delta\vec{a}_i
\end{aligned}
$$

   Les paramètres $\alpha_0, \alpha_1, \ldots, \alpha_5$ sont fonction du degré des équations différentielles et de l'ordre du développement en série de Taylor, ils assurent la stabilité numérique des solutions du système.

Conditions initiales

Pour amorcer l'algorithme de prédiction-correction, les valeurs des positions r_i et leurs cinq premières dérivées[6] doivent être imposées à l'instant $t = 0$. Ces valeurs sont en général appelées les conditions initiales.

Dans notre simulation, les positions des particules sont simplement les noeuds d'un réseau périodique.
Les composantes de la vitesse proviennent d'une distribution de Gauss et doivent obéir à la relation de Maxwell-Boltzmann, liant une température T[7] et la probabilité $p(\vec{v_i})$ qu'une molécule de masse m_i possède une vitesse $\vec{v_i}$. Cette procédure est connue sous la thermalisation initiale. En plus, les vitesses des atomes doivent vérifier que le centre de masse du système est au repos.[8]
Aussi l'accélération est initialisée via une prédiction à partir des positions initiales.
Les valeurs initiales des dérivées d'ordre supérieur sont assignées à la valeur zéro, le système s'équilibre automatiquement après quelques centaines d'itérations.

Conditions au bord et portée du potentiel

Deux facteurs importants pour le calcul des forces d'interaction (étape d'évaluation) sont le principe des conditions aux bords périodiques et la portée du potentiel.

En simulation numérique la taille des systèmes est limitée par la place de stockage disponible sur l'ordinateur et de manière plus cruciale par la vitesse d'exécution du programme. En général, on emprisonne les particules d'une simulation dans une cellule. Alors les particules qui sont situées à proximité d'une face de la cellule sont soumises à des forces totalement différentes de celles supportées par les particules au coeur du système. Ce problème peut être écarté par l'implémentation des conditions aux bords périodiques [7]. La cellule est reproduite autour d'elle-même dans l'espace, pour former ainsi un réseau virtuel infini.

Le calcul du potentiel U demande le calcul des potentiels d'interaction[9] entre chaque paire d'atomes de la cellule initiale, mais aussi des potentiels entre un atome de la cellule initiale et les atomes des cellules du bord. Pour l'étude de grand système, cette technique est inappropriée, puisque le calcul d'un grand nombre de termes à chaque itération consomme du temps de calcul. De plus, la majeure contribution au potentiel, donc à la force, provient des plus proches voisins de la particule concernée. C'est pourquoi la portée des potentiels est habituellement tronquée. Seuls les particules j contenues dans une sphère de rayon r_c centrée sur la particule i interviendront dans le calcul de la force exercée sur i. On appelle r_c le rayon de coupure. L'utilisation du rayon de coupure permet un gain de temps considérable mais ne peut induire qu'une faible perturbation du système, il convient donc de choisir une bonne valeur pour le rayon de coupure selon la nature réel du système.

[6] Comme au-dessus, ce sont la vitesse, l'accélération ainsi que les dérivées troisième, quatrième et cinquième de la position.
[7] La température est choisie au préalable.
[8] Le centre de masse d'un système à l'équilibre, sans perturbation est au repos.
[9] Donc le potentiel de Lennard-Jones et le potentiel harmonique(intramoléculaire).

Équilibre, calcul des propriétés et fin de la simulation

Au cours des premières centaines de pas de temps, le système évolue depuis sa configuration initiale vers un état d'équilibre. Généralement la température du système dévie par rapport à sa valeur prescrite. Pour maintenir le système à cette température cible, de l'énergie sous forme de vitesses est artificiellement injectée ou soustraite. Après que cette étape de thermalisation artificielle est terminée le système est à équilibre et peut être exploité pour le calcul de propriétés. Il convient de s'assurer que la propriété observée reste stable dans le temps avant d'interrompre la simulation.

5.2 Description de la simulation

Dans le développement de nouveaux médicaments qui agissent directement au niveau des molécules la manière dont une membrane de lipides s'étale sur un substrat solide joue un rôle important. La simulation numérique empruntée de ce chapitre a pour objectif d'étudier cet étalement.
En pratique, la simulation a été réalisée dans le cadre du travail de fin d'étude [3] d'Emilie Bertrand, une étudiante en licence en science physique à l'université de Mons-Hainaut (UMH), et a été développée en collaboration avec le CRMM (Centre de Recherche en Modélisation Moléculaire) du pôle d'excellence "Materia Nova", situé également à Mons.

Le but de la simulation est de modéliser des films de Langmuir-Blodgett (figure 5.2) obtenus expérimentalement en déposant des molécules lipidiques sur une surface d'eau. Des barrières en Téflon permettent, en se rapprochant, d'obtenir une monocouche. La pression au sein de celle-ci est choisie en modifiant la distance entre les barres de Téflon. Ensuite, il est alors aisé de transférer la monocouche sur un substrat solide en maintenant la valeur de la pression latérale.

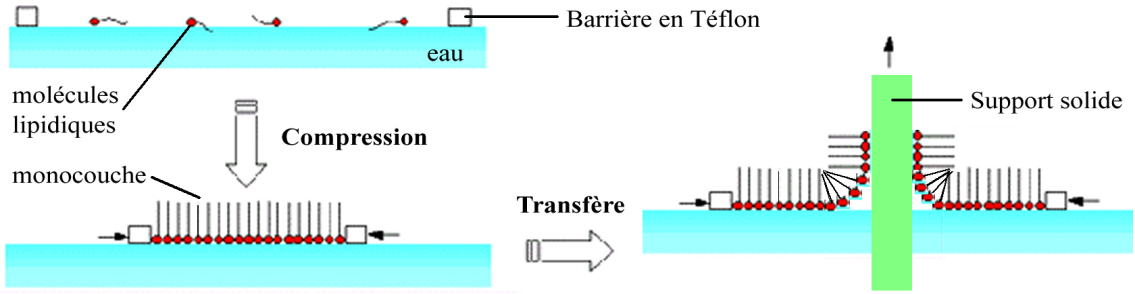


FIG. 5.2 – *Fabrication expérimentale d'une monocouche par la méthode de Langmuir-Blodgett*

La simulation est composée de deux parties. Une partie préparatoire qui laisse la monocouche s'équilibrer sur le substrat solide en vérifiant la dépendance en 1/aire par molécule pour la pression latérale. Seule la deuxième partie analyse l'étalement de la monocouche sur le substrat solide en considérant la valeur de la pression calculée pendant la première partie. En effet, des théories physiques et des observations expérimentales prédisent que la pression latérale est en fonction non-linéaire avec le taux d'étalement des monocouches de Langmuir-Blodgett. Cette simulation essaie de valider ce mécanisme.
Il est clair, que la deuxième partie de la simulation propose un comportement beaucoup plus dynamique que la première et donc aussi plus favorable pour des gains de performance via un équilibrage dynamique de charge pendant la résolution numérique. Malheureusement au moment du stage, seule la première partie de la simulation était disponible. Nous

allons donc essayer de voir des avantage pour l'équilibrage dynamique avec une application avec une exécution assez statique. Pour voir des résultats nous sommes alors obligés d'introduire un déséquilibre externe. Dans une perspective de ce mémoire il serait intéressant de refaire les tests avec la deuxième partie plus dynamique de la simulation.
Notre programme calcule donc une propriété du système, à savoir la pression latérale. Pour différentes valeurs de l'aire par molécule, nous laissons s'équilibrer le système et nous déterminons la pression avec le tenseur de pression.

Nous décrivons maintenant la modélisation (figure 5.3) de la monocouche et du substrat solide ainsi que des potentiels pris en considération.

5.2.1 La monocouche

La monocouche est constituée de longues chaînes placées les unes à côté des autres, verticalement. Elle peut donc être vue comme un alignement de chaînes représentées précédemment (figure 5.2). Les atomes la constituant sont ceux d'argon[10]
Les interactions intervenant sont :

1. Le potentiel de Lennard-Jones entre chaque paire d'atomes (aussi du substrat solide)
2. Un potentiel harmonique (intramoléculaire) entre un atome et ses premiers voisins dans une même chaîne pour que celle-ci garde son aspect.
3. Un potentiel entre la première atome d'une chaîne et les atomes du solide, pour s'assurer que les chaînes restent fixées au substrat solide. Aussi le fait qu'on a transféré la monocouche sur le substrat solide, donne lieu a une interaction différente entre la base d'une chaîne et le solide.

5.2.2 Le substrat solide

Le substrat solide est formé d'un réseau plan carré. Les atomes le constituant sont ceux de carbone. Chaque atome est placé à un noeud du réseau, position qui sera sa position d'équilibre.
Les interactions intervenant sont :

1. Le potentiel de Lennard-Jones entre chaque paire d'atomes (aussi de la monocouche)
2. Un potentiel harmonique pour chaque atome autour de sa position d'équilibre pour garder l'aspect d'un solide.

5.3 Conception du programme parallèle

Le programme [3] de la simulation était sous forme de code C et souffrait des temps de calcul trop importants pour faire des tests réels. D'où l'idée de le paralléliser en code MPI.

La parallélisation d'un code séquentiel n'est pas une tâche facile. D'abord le programme, étant écrit par une non-informaticienne, était peu structuré. En suivant la méthodologie de la dynamique moléculaire (figure 5.1), l'introduction de quelques fonctions et structures de données rend le programme plus lisible et plus facilement transformable. Observons maintenant (figure 5.4) le pourcentage des temps d'exécution des différentes étapes d'une itération du programme séquentielle.

[10]Le choix pour les atomes d'argon permet de garder un aspect général. En fait, ce qui caractérise le fait que ce soient des atomes d'argon, c'est la nécessité de fixer la masse et certaines valeurs physiques pour le calcul du potentiel de Lennard-Jones.

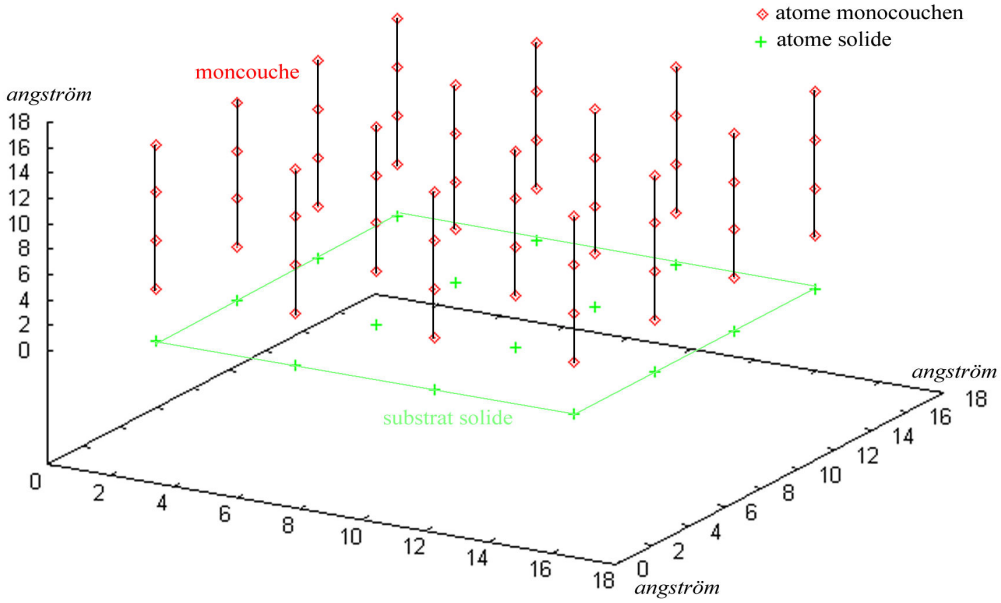


FIG. 5.3 – *Modélisation de la monocouche et du substrat solide à l'état initiale*

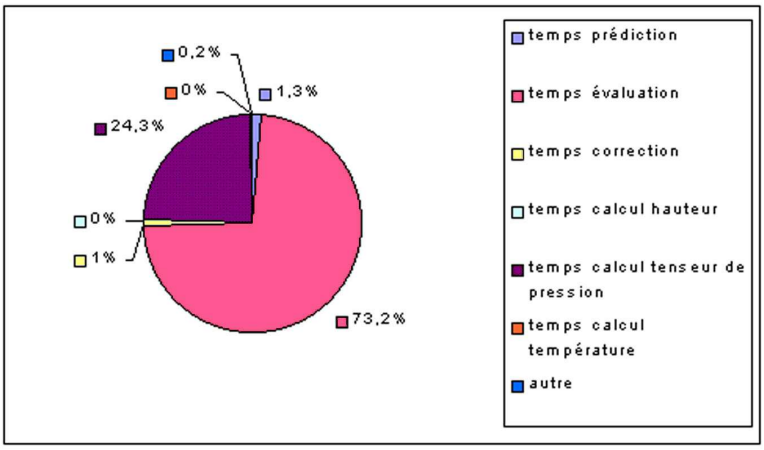


FIG. 5.4 – *Pourcentage des temps d'exécution des différentes étapes d'une itération (Les pourcentages sont calculé à partir des valeurs se trouvant à l'annexe A.2)*

L'étape d'évaluation, donc le calcul des forces d'interactions, consomme 75% de temps de calcul. Le pourcentage augmente encore si on considère que le tenseur de pression est seulement calculé dans les itérations après la phase de thermalisation artificielle. Dans les tests effectués pour ce mémoire cela revient au dernier tièrs des itérations. Les itérations préalables à l'étape d'évaluation prennent alors jusqu'à 95% de temps de calcul. Il convient donc de paralléliser uniquement cette étape de calcul des forces d'interaction.
Comme l'étape des accélérations initiales demande aussi le calcul des interactions et utilise les mêmes fonctions que la partie parallélisée, nous avons également décidé de la paralléliser.
Pour une raison de conception nous avons choisi un modèle de maître-esclave (voir aussi page 61). Le maître prend en charge toutes les étapes non parallélisées du programme. Aussi à chaque itération il va partager les données de la simulation en différentes boîtes pour les distribuer aux esclaves. Nous avons donc un partage en blocs des données. Chaque esclave calcule alors les interactions des paires d'atomes concernant sa boîte. L'organi-

gramme de dynamique moléculaire modifié est montré à la figure 5.5.

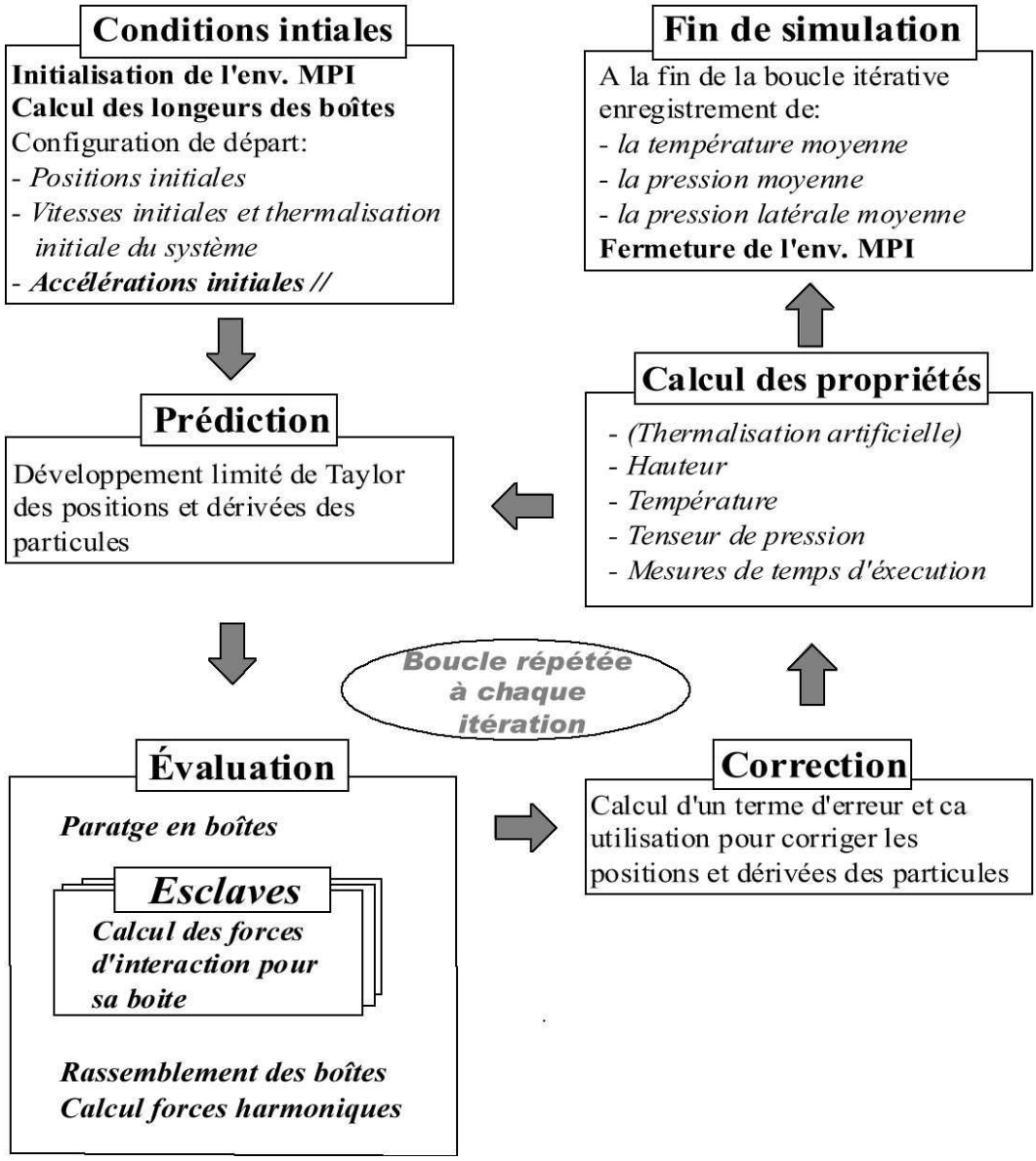


FIG. 5.5 – *Organigramme du code parallèle*
(Les termes en gras marquent les adaptations pour la version parallèle)

Pour que l'esclave puisse aussi calculer les forces d'interaction totales des atomes qui se trouvent au bord de sa boîte, le maître passe en plus une boîte "bord" avec les atomes qui se trouvent avec une distance plus petite que le rayon de coupure du bord de la boîte de l'esclave (voir figure 5.6). Ainsi nous pouvons aussi simuler les conditions aux bords périodiques (voir page 76).

Nous pouvons résumer la partie parallèle entre le maître et les esclaves dans la figure 5.7.

Remarquons encore que le maître attend les esclaves pendant leurs calculs et l'inverse que les esclaves attendent quand le maître calcule. Dans MPI cette constellation n'est

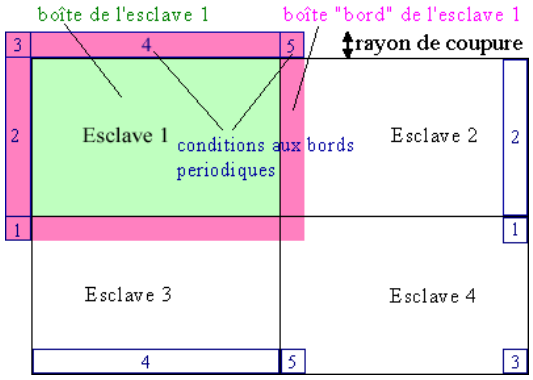


FIG. 5.6 – *Division en boîtes pour quatre esclaves avec la boîte (en vert) et la boîte "bord" (en rouge) de l'esclave 1. Pour les conditions aux bords périodiques on copie les rectangles 1 à 5 (en bleu)*
(Vue en 2 dimensions, même principe pour la simulation réelle en 3 dimensions)

Maître		**Esclave**
Division en boîtes		
Envoi des boîtes	→	Réception de la boîte
		Calcul des forces d'interaction
Réception des boîtes résultats	←	Envoi de la boîte résultat
Concaténation des boîtes		

FIG. 5.7 – *Partie parallèle entre le maître et les esclaves*

pas idéale car on distribue en général un processus, soit un maître soit un esclave, par processeur. Donc il y a toujours au moins un processeur qui attend. Avec le principe de virtualisation des processeurs l'environnement AMPI peut occuper théoriquement plus équitablement tous les processeurs en mettant le processeur virtuel "maître" avec des processeurs virtuels "esclave" sur un même processeur réel.

5.4 Documentation du programme *press*

D'abord notons qu'il existe 5 versions différentes du programme *press*, à savoir : le programme séquentiel d'origine en C [3], sa version séquentielle structurée, une version séquentielle qui est implémentée avec les mêmes sous-routines que les versions parallélisées mais seulement exécutable sur une seule machine, et finalement les version parallélisées en MPI avec les appels du *message-passing* et AMPI avec les possibilités de migration.
A l'exception du programme d'origine toutes les versions[11] ont la même structuration avec des petites adaptations dans le code des sous-routines. Comme l'aspect parallèle de l'application nous intéresse le plus, nous allons nous limiter pour la documentation aux versions parallèles en spécifiant les différences entre la version MPI et la version AMPI.

5.4.1 Pré-requis

Les versions ont été développées pour des systèmes exploitations Linux. Spécialement ils ont été testés pour *RedHat 8.0*. Normalement un système disposant un compilateur C

[11] Les quatre versions ont été développées et implémentées durant le stage. (voir aussi `http://users.skynet.be/teller/memoire/`)

de la gamme *gcc* est suffisant pour les versions séquentielles.
La version en MPI nécessite une implémentation du standart MPI. Nous suggérons l'implémentation freeware *MPICH*.
Pour la version AMPI une installation de l'environnement AMPI de CHARM++ est nécessaire.

5.4.2 Les paramètres du programme

L'application a besoin de 17 paramètres obligatoires qui sont passés au lancement via ligne de commande. A ces paramètres s'ajoutent des options spécifiques à l'environnement utilisé.
Dans l'ordre, les paramètres sont :

1. **nbrAtomeCoteSol** : le nombre d'atomes qui forment un coté du solide.
   Le carré de `nbrAtomeCoteSol` donne alors le nombre d'atomes du substrat solide.
2. **distAtomeSol** : la distance entre les atomes du solide en unités "sigma". Cette valeur représente le diamètre des atomes dans le solide. Elle intervient dans le potentiel de Lennard-Jones. En général, ce paramètre prend la valeur 1.
3. **nbrAtomeChaineMem** : le nombre d'atomes qui forment une chaîne de la monocouche.
4. **nbrChaineCoteMem** : nombre de chaines qui forment un coté de la monocouche. `nbrChaineCoteMem` ^{2} `* nbrAtomeChaineMem` donne alors le nombre d'atomes de la monocouche.
5. **tempRef** : la température de reférence en Kelvin. En général cette valeur est fixée à 300K.
6. **rayonCoupure** : la valeur du rayon de coupure du potentiel en Angström. En général on prend 15 Angström.
7. **rapSol** : une constante de rappel dans le solide en Newton par Angström. Cette valeur est utilisée pour calculer le potentiel harmonique tel que chaque atome du solide reste autour de sa position d'équilibre afin de garder l'aspect d'un solide. Pour les tests la valeur 1000 a été prise.
8. **rapMem** : une constante de rappel dans la monocouche en Newton par Angström. Cette valeur est utilisée pour calculer le potentiel harmonique entre un atome de la monocouche et ses premiers voisins dans une même chaîne pour que ceux-ci gardent leur aspect. Pour les tests la valeur 5 a été prise.
9. **rapBaseChaine** : une constante de rappel à la base de la chaîne en Newton par Angström. Cette valeur est utilisée pour calculer un potentiel entre le premier atome d'une chaîne et les atomes du solide, pour s'assurer que les chaînes restent fixées au substrat solide. Pour les tests la valeur 100 a été prise.
10. **nbrIterEquilibre** : le nombre d'itérations que le programme effectue.
11. **nbrIterScaling** : le nombre d'itérations pour la thermalisation artificielle.
12. **nbrIterAttente** : le nombre d'itérations d'attente entre la thermalisation et l'enregistrement des mesures fait par le tenseur de pression. On doit garder que `nbrIterEquilibre > nbrIterScaling + nbrIterAttente`.
13. **initRand** : constante entière d'initialisation de la valeur aléatoire.
14. **nbrBtX** : le nombre boîte selon l'axe X.
15. **nbrBtY** : le nombre boîte selon l'axe Y.
16. **nbrBtZ** : le nombre boîte selon l'axe Z. `nbrBtX * nbrBtY * nbrBtZ` est le nombre total des boîtes qui doit correspondre au nombre d'esclaves lancés.

17. **repertoire** : le répertoire pour les fichiers résultats.

Pour la version en AMPI il s'ajoute un paramètre :

18. **periodeLB** : le nombre d'itérations entre deux tests de migration.

Les options spécifiques pour l'environnement MPI sont par exemple :

- **-np x** : où x donne le nombre de processeurs sur lequel l'application devra s'exécuter. Par exemple, si x est 5, ça veut dire qu'on a un maître et quatre esclaves. Au minimum x vaut 2 car le programme a besoin d'au moins un maître et un esclave.
- **-machinefile fichier** où fichier est un fichier texte qui contient les adresses IP (ou les aliases) des machines qui participent au calcul.

Pour l'environnement AMPI ces options deviennent :

- +**p x** : où x donne le nombre de processeurs réels sur lequel l'application devra tourner. Au minimum x vaut 2.
- +**vp y** : où y donne le nombre de processeurs virtuels sur lequel l'application devra tourner. Par exemple si x est 5 et y est 10, ça veut dire qu'on a un maître et neuf esclaves qui s'exécutent sur 5 processeurs réels.
- +**balancer strategieLB** : où `strategieLB` spécifie la stratégie qui est utilisée pour l'équilibrage dynamique.
- un fichier `.nodelist` doit se trouver dans le répertoire principal de l'utilisateur qui lance l'application et qui spécifie les adresses IP des machines participantes.

5.4.3 Les fichiers résultats

Le programme *press* génère une série de fichiers résultats :

- **confm1.dat** : contient les coordonnées initiales x, y et z (en trois colonnes) des atomes de la monocouche.
- **confm2.dat** : contient les coordonnées finales x, y et z (en trois colonnes) des atomes de la monocouche.
- **confs1.dat** : contient les coordonnées initiales x, y et z (en trois colonnes) des atomes du substrat solide.
- **confs2.dat** : contient les coordonnées finales x, y et z (en trois colonnes) des atomes du substrat solide.
- **hautmoy.dat** : contient la hauteur moyennes instantanée des chaînes au cours de la simulation ainsi que la grandeur moyenne des hauteurs moyenne sur le temps.
- **tempmem.dat** : contient la température instantanée de la monocouche au cours de la simulation
- **tempsol.dat** : idem mais pour le solide
- **tempmoy.dat** : contient la température moyenne de la monocouche au cours de la simulation
- **pressxx.dat** : contient la valeur instantanée et moyenne de la composante xx du tenseur de pression
- **pressyy.dat** : idem mais pour la composante yy
- **presszz.dat** : idem mais pour la composante zz
- **pressinst.dat** : contient la valeur instantanée de la pression obtenue avec la trace du tenseur de pression et la pression latérale instantanée obtenue de la même façon

– **pressmoy.dat** : idem mais pour la valeur moyenne au cours du temps

– **temps.dat** : contient les mesures de temps en msec des itérations ainsi les temps de chaque étape des itérations.

– **resultat.dat** : contient les valeurs finales de la simulation : la hauteur moyenne, la pression moyenne et la pression latérale moyenne ; ainsi que les écarts types de ces valeurs.

5.4.4 Le code

Le code du programme est distribué (figure 5.8) dans 9 fichiers sources et 2 fichiers déclaratifs(.h) différents.
Le fichier principal s'appelle `pressXPar1.c`, où le *X* représente la version par exemple MPI ou AMPI. Il contient la fonction *main* du programme, qui est la représentation en langage C de l'organigramme de la dynamique moléculaire (figure 5.5). Son fichier `pressXPar.h` fixe les constantes du programme et définit les structures de données utilisées.
Le fichier principal inclut un fichier `para.c` qui avec l'aide du `parametre.h` déclare toutes les variables globales du programme. Le fichier `parametre.c` met aussi à disposition des fonctions de traitement de ces variables, par exemple l'initialisation avec les paramètres de l'appel du programme.
Le deuxième fichier, auquel le fichier principal fait appel, est le fichier `etapeDynMol.c`. Il met à disposition l'implémentation en sous-routines des différentes étapes de la figure 5.5. Les sous-routines de `etapeDynMol.c` font appel à des fonctions qui sont dans autres fichiers sources. Notamment à `CondInit.c` qui implémente les sous-routines pour les conditions initiales, à `accInterHelper.c` qui fournit de même les sous-routines pour calculer les accélérations, donc les forces, dues à l'interaction, et à `calcFin` qui contient les sous-routines pour l'enregistrement à la fin du programme.
Un fichier `fonctPhys.c` met à disposition des fonctions qui calculent les valeurs physiques comme le potentiel de Lennard-Jones. Le fichier `mySys.c` propose des fonctions systèmes adaptées, par exemple pour l'ouverture des fichiers, et `affiche.c` des fonctions pour affiché les structures de données surtout utiles pour le débuggage.

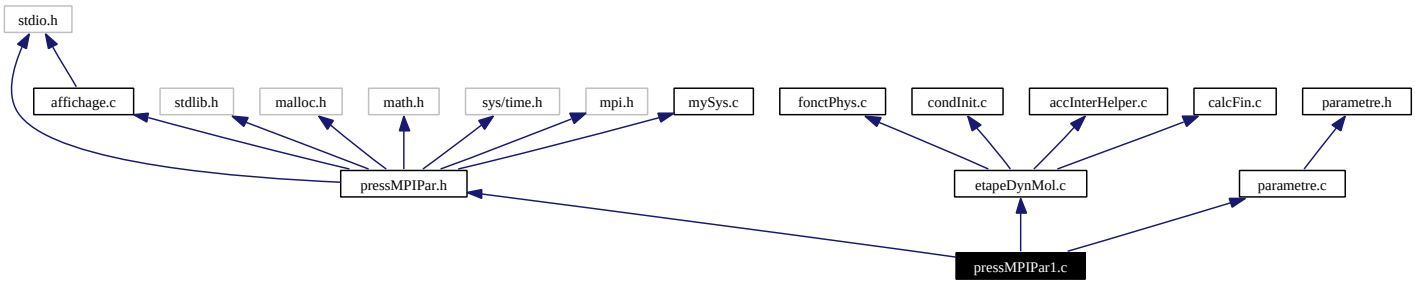


FIG. 5.8 – *Graphes des dépendances du programme* MPI.

Notons encore que le programme est le même pour le maître et pour l'esclave en MPI (modèle SPMD). Seuls des tests sur le rang du processus, fixé par l'environnement MPI, permettent de renvoyer ou de sauter différentes sous-routines qui simulent alors le comportement, soit du maître, soit de l'esclave. La différence entre maître et esclave est gérée dans la fonction *main*.

5.5 Résultats et interprétation des tests

Dans cette section nous allons faire des comparaisons entre les résultats des différentes versions (voir page 81) du programme *press*. Notamment, les fichiers résultats sont comparés, des tests sur le temps de calcul sont présentés et quelques remarques sur l'équilibrage dynamique de charge dans cette application pratique. Ainsi, le passage du code MPI à du code AMPI est analysé.

Les jeux de tests utilisés jouent sur le nombre d'atomes qui constituent la simulation. Les paramètres exacts peuvent être trouvés dans l'annexe A.1 à la page 103.
Notons encore que tous les tests ont été effectués sur l'architecture cible du mémoire et était la même que pour le programme *sum*, plus précisément un réseau de PC Linux avec des processeurs Intel Pentium 4 de 2.00 Ghz et 256 Mo de Ram. Les ordinateurs étaient connectés via un Ethernet 100 Mb/s. Vu que le débit de ce réseau n'est pas le meilleur à l'heure actuelle, les versions parallélisées sont désavantagées car alors le temps de communication peut prendre de l'importance par rapport au temps de calcul total.

5.5.1 Comparaisons des fichiers résultats

Nous comparons ici pour le jeu de test 7 les valeurs de quelques fichiers résultats de différentes versions de l'application pratique. Le premier graphique (figure 5.9) montre l'évolution des pressions moyennes calculées par le tenseur de pression qui commence son calcul à partir de la 20 000^{e} itérations.

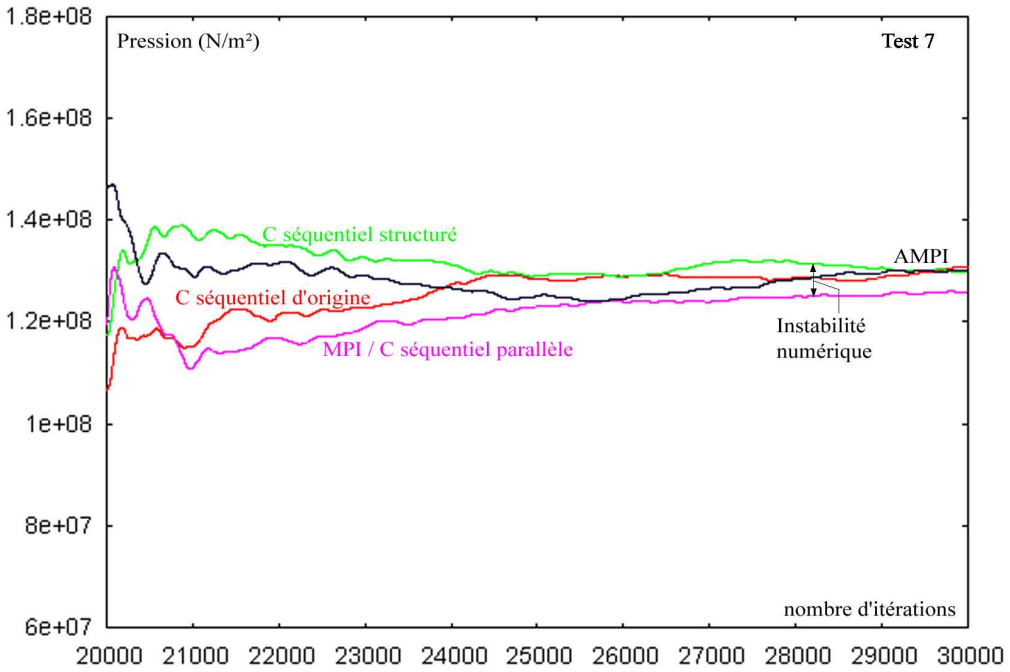


FIG. 5.9 – *Valeurs de la pression moyenne calculées par le tenseur de pression pour le jeux de test 7*

Le deuxième graphique (figure 5.10) montre l'évolution des hauteurs moyennes de la monocouche sur 30 000 itérations.

Nous remarquons dans les deux graphiques que les différentes versions ne donnent pas les mêmes résultats. Or les conditions initiales sont identiques et la résolution numérique

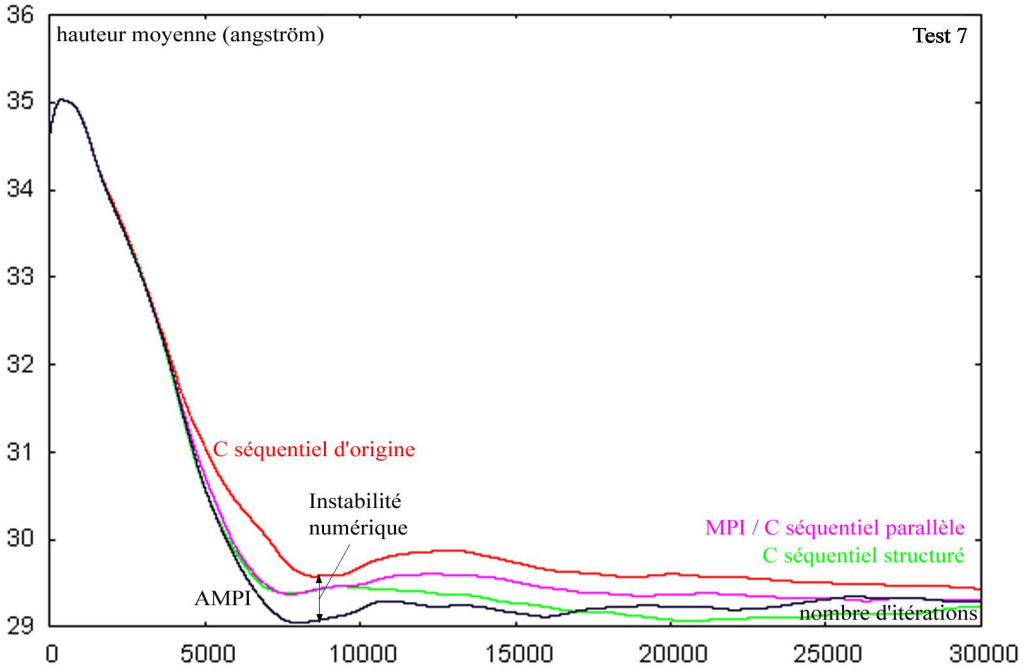


FIG. 5.10 – *Valeurs de la hauteur moyenne pour le jeux de test 7*

utilisée est déterministe. Ces différences sont dues à des erreurs d'arrondi. En fait, le programme calcule avec des nombres d'ordre de grandeur différent, ce qui induit des erreurs d'arrondi.
Précisons l'origine de ces erreurs, le programme travaille avec des données du type *double*, donc seulement les 16 premiers chiffres sont significatifs ; tous les chiffres qui viennent après sont aléatoires. Comme la suite des additions des forces d'interactions pour un atome change d'une version du programme *press* à l'autre les résultats finals d'ordre de grandeur plus petit que les opérants de ces opérations commutatives en mathématique diffèrent. En effet, dans la simulation du programme un atome est entouré par d'autres atomes tel que la somme des forces d'interaction le garde plus ou moins à sa place. Aussi l'entourage n'est pas tout à fait symétrique, ainsi les forces ne s'annulent pas complètement. Pour un atome les différentes forces d'interaction sont donc d'un genre opposé, ainsi les résultats finals sont en général de puissance plus petite que les opérants. Les chiffres avant non significatifs prennent alors des places significatives et sont considérés pour des prochains calculs[12].
Pour visualiser cela, observons le graphique de la figure 5.11 qui montre les courbes des hauteurs moyennes de deux programmes identiques, sauf qu'on s'est arrangé que l'ordre, selon lequel les forces d'interaction sont additionnées, diffère. En comparant les résultats du programme original et de sa version modifiée, nous constatons des fluctuations semblables à celles des deux graphiques présentés avant.
La modification de l'ordre d'addition a été implémenté en changeant au début d'un programme la numérotation des atomes, par exemple l'atome numéro 1 devient l'atome numéro 9, l'atome 9 devient l'atome 1, ... Alors les conditions initiales de la simulation sont toujours identiques à celle du programme original, mais l'ordre de traitement des atomes, donc l'ordre dans lequel nous additionnons les différentes forces d'interaction par atome, n'est plus la même[13].

[12] Un exemple d'une erreur d'arrondis se trouve à l'annexe C
[13] Pour respecter certains calculs, notamment pour le calcul des forces harmoniques dans une chaîne et

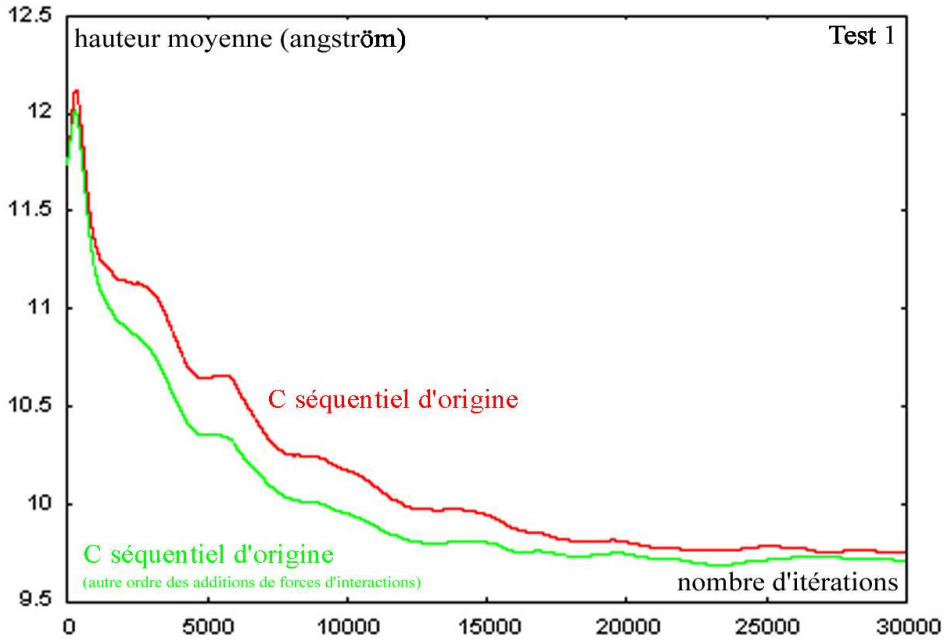


FIG. 5.11 – *Comparaison pour les courbes des hauteurs moyennes de la monocouche entre deux programmes séquentiels d'origine, différente uniquement dans l'ordre ou l'addition des forces d'interactions se fait, pour le jeu de test 1*

Dans notre simulation, le temps que les erreurs d'arrondi se propagent et s'agrandissent fait que les courbes dans la figure 5.10 sont fort semblables les 1000 premières itérations et commencent de s'éloigner après. Dans le premier graphique (figure 5.9) nous ne remarquons pas ce phénomène car les enregistrements de la pression débutent seulement à la 20 000^{e} itération quand les erreurs se sont déjà bien propagées. Les fluctuations de début de ce graphique sont dues au fait que la moyenne ne comprend pas encore assez de valeurs instantanées pour être représentatives.
Néanmoins, les deux graphiques convergent plus ou moins vers des valeurs uniques[14] et des chercheurs de *Materia Nova* ont confirmé que de telles instabilités numériques pouvaient être possibles et tolérables.

5.5.2 Les temps de calcul

Regardons d'abord le tableau suivant (5.12 et figure 5.13) qui montre les temps d'exécution des différentes versions du programme *press* pour différents jeux de test.
La version MPI a été exécutée sur 4 processeurs, de même la version AMPI, mais qui lance sur ces 4 processeurs 19 processeurs virtuels. En plus les tests avec la version AMPI utilisent la stratégie WSLB pour un réseau de stations de travail et font un test de migration chaque 1 000^{e} itération.

pour l'identification des atomes qui constituent une base ou la fin d'une chaîne, on garde que les atomes d'une chaîne de la monocouche ont encore après la permutation des numéros successifs croissants. Pour les atomes du solide cette restriction n'est pas obligatoire.

[14]Les erreurs d'arrondi sont corrigées par des réactions du système qui tente d'aller vers un équilibre et de plus s'annulent en moyenne à cause de leur distribution aléatoire et uniforme.

	Prog. C séquentiel d'origine	**Prog. C séquentiel structuré**	**Prog. C séquentiel paralisé**	**Prog. MPI parallélisé**	**Prog. AMPI parallélisé**
test 1	**00 :01 :14**	00 :01 :55	00 :01 :57	00 :01 :52	00 :05 :42
test 2	00 :08 :22	00 :10 :27	00 :12 :10	**00 :06 :18**	00 :11 :25
test 3	00 :29 :32	00 :33 :32	00 :34 :19	**00 :17 :41**	00 :22 :09
test 4	01 :22 :54	01 :30 :22	01 :32 :23	**00 :42 :39**	00 :47 :03
test 5	03 :57 :14	04 :12 :05	04 :17 :51	**01 :54 :16**	01 :58 :16
test 6	06 :18 :13	06 :35 :00	06 :45 :04	**02 :57 :15**	03 :00 :13
test 7	09 :31 :07	09 :52 :00	09 :56 :07	**04 :19 :00**	04 :24 :11
test 8	13 :49 :44	14 :12 :36	14 :15 :54	06 :12 :31	**06 :03 :54**

FIG. 5.12 – *Temps d'exécution des différentes versions du programme press pour différents jeux de tests*

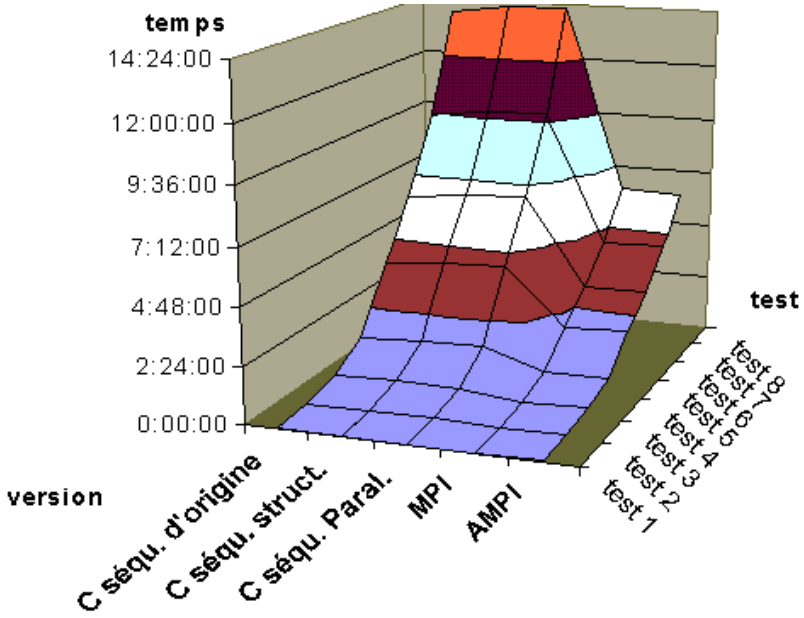


FIG. 5.13 – *Vue nappe sur les temps d'exécution*

D'abord nous remarquons dans le tableau que le programme séquentiel structuré est plus lent que le programme séquentiel d'origine. Cela est dû au sur-coût introduit avec la découpe en fonction et le parcours des structures de données introduites. La version séquentielle parallélisée a encore un sur-coût qui est causé par les fonctions de parallélisation comme la division en boîte.

En général, la version MPI est la plus performante pour le temps d'exécution. La figure 5.14 montre le speedup[15]entre la version séquentielle parallélisée et le programme en MPI.

[15]speedup = temps du *meilleur* programme séquentiel / temps du programme parallélisé exécuté sur n processeurs
(Ici le programme séquentiel qui correspond à la réalité physique du programme parallèle a été prise.)

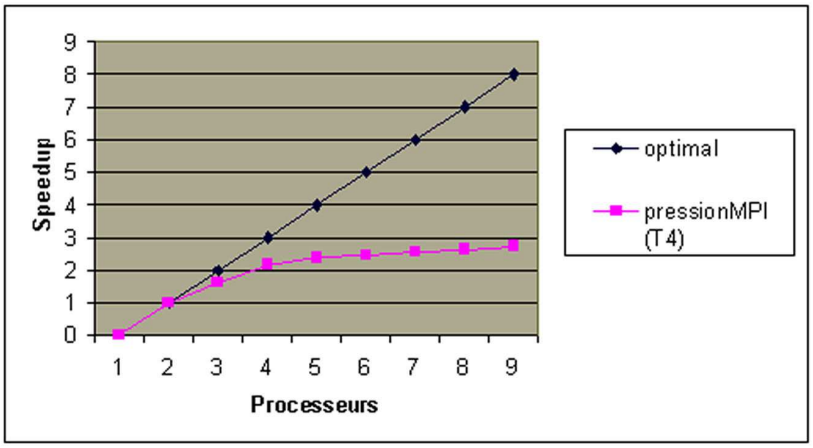


FIG. 5.14 – *Speedup entre la version séquentielle parallélisée et le programme en* MPI

Le speedup est loin d'être optimal. D'abord c'est dû au fait que le programme est seulement parallélisé à 75%(voir page 78), donc 25% du programme sont toujours éxecutés en séquentiel sur un seul processeur. Cela implique que le speedup théorique ne peut pas aller au delà de 4. Puis, l'architecture utilisée (voir page 85), surtout la lenteur du réseau est un frein à des meilleurs speedup à cause des coûts de communication trop importants si le nombre de processeurs augmente.

La version AMPI est en général plus lente que la version MPI. Cela est dû à la nature trop statique de l'application qui rend presque nuls les gains qu'on pouvait avoir avec des migrations éventuelles. En effet, pendant les tests le nombre de migrations réelles n'a jamais dépassé 4. Par contre, nous avons un sur-coût pour la version AMPI du au test de migration et l'application de la stratégie de l'équilibrage de charges.
Notons, que pour le test 8 AMPI montre un avantage par rapport à MPI. Cela vient des profits du concept de virtualisation des processeurs (voir figure 5.15). Nous supposons que, dans les tests 1 à 7 le coût de communication avec les 18 esclaves (en MPI seulement 3) était trop important par rapport au temps de calcul des esclaves et comblait les gains dus à la virtualisation des processeurs.

Sur le graphique (figure 5.16) nous pouvons examiner plus précisément pour le test 8 les temps d'itération des différentes version du programme *press*.
D'abord le saut dans les courbes montre le début du calcul du tenseur de pression à l'itération 20 000. Celui est entièrement calculé par le processeur "maître".
Puis toutes les petites barres pour la version AMPI montrent des tests de migration chaque mille itérations. Il n'y a pas nécessairement une migration à ces moments. En effet en réalité, il y avait seulement des migrations à l'itération 4 000 et 21 000. Pour ces deux itérations nous remarquons que pour des migrations le temps est un peu plus long qu'aux tests de migration sans aucune migration.
La première itération est positive et apporte un gain de temps. En fait c'est une petite adaptation par la stratégie d'équilibrage des charges utilisées, à savoir la stratégie WS (*Workstation Load Balancer*), avec un gain minime.
La deuxième migration à l'itération 21 000 dégrade les performances en temps de calcul. L'origine de la migration est le début du calcul du tenseur de pression qui va totalement à la charge du processeur maître. Le fait que la migration est négative provient d'une interprétation fausse de la stratégie d'équilibrage de charges sur la nature du programme. Rappelons-nous la figure 5.15 où la première migration a abouti dans un passage du l'es-

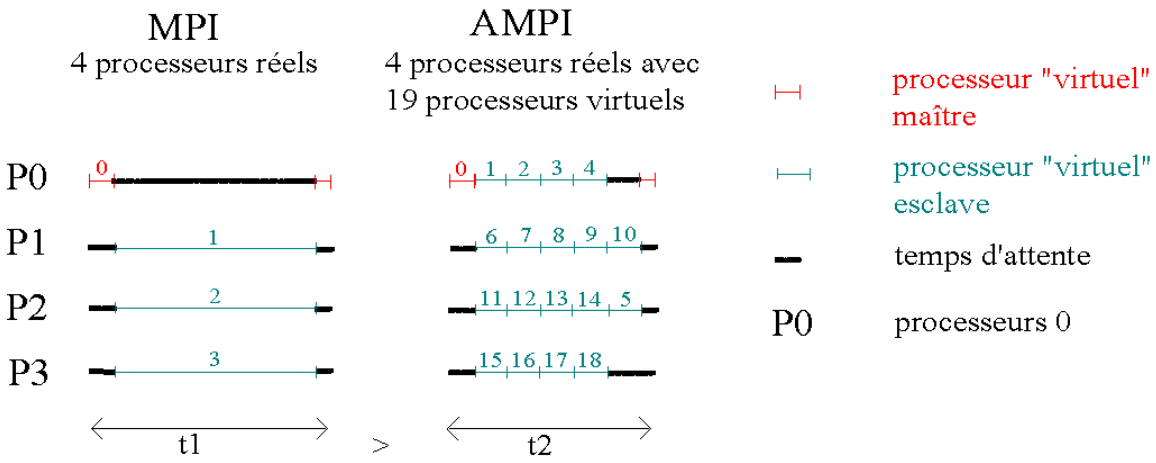


FIG. 5.15 – *Application du concept de virtualisation des processeurs de* AMPI *par rapport à* MPI *pour une itération du programme* press

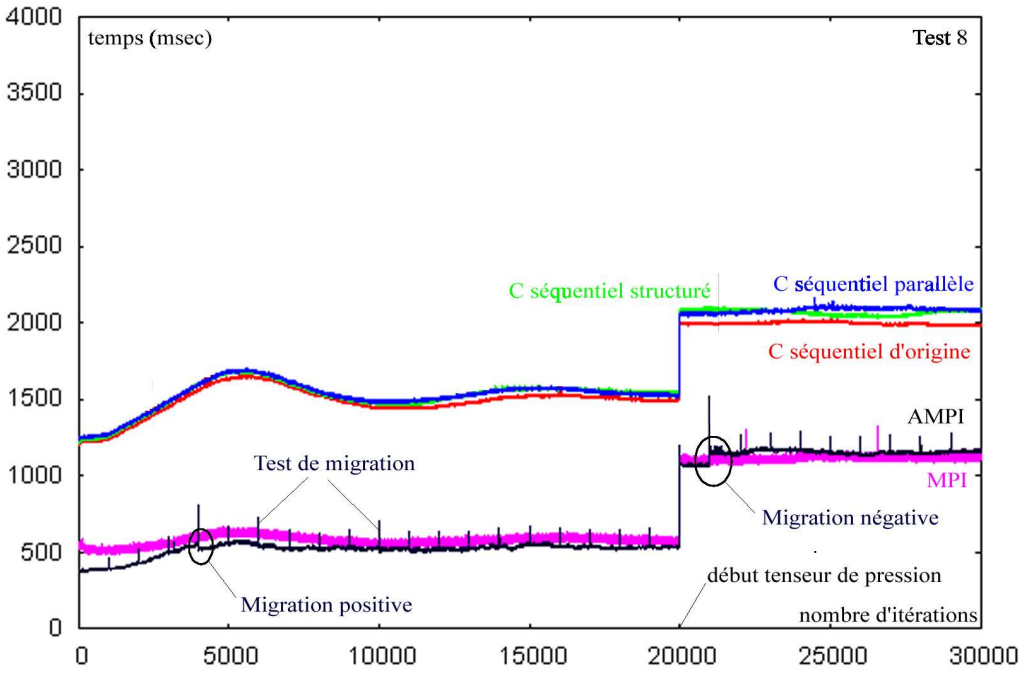


FIG. 5.16 – *Temps des itérations pour les 5 versions du programme* press*, pour le jeu de test 8*

clave 1 de $P0$ à $P3$. Ajoutons pour le maître la charge due au calcul du tenseur de pression. Dans la suite des étapes d'une itération le tenseur de pression est calculé à la fin de l'itération, donc après que les esclaves ont fini leurs calculs (figure 5.17 gauche).
La stratégie d'équilibrage de charges détecte correctement que le processeur $P0$ à maintenant plus de charge que les trois autres processeurs. Il va donc migrer les trois esclaves qui résident sur $P0$ vers les autres processeurs. Mais la stratégie ne détecte pas que les esclaves et le maître ne peuvent pas s'exécuter simultanément. Donc, après la migration, le temps d'attente du maître est six fois le temps de calcul d'un esclave par rapport à cinq

fois le temps de calcul avant (figure 5.17 droite). La charge est donc mieux équilibrée, mais le gain de la virtualisation est perdu à cause d'un effet de synchronisation.

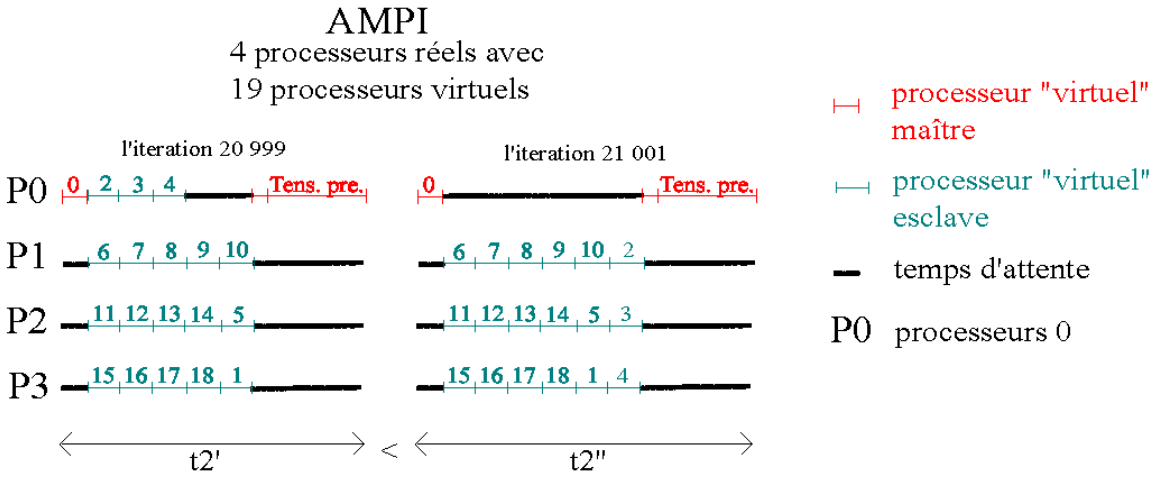


FIG. 5.17 – *Migration à l'itération 21 000*

Remarquons encore, comme les stratégies d'équilibrage de charges sont inclues en module à AMPI, il est possible décrire sa propre stratégie pour tenir compte de la nature de synchronisation de l'application et de son développement.

5.5.3 Résultats avec charges extérieures

Pour la stratégie WS, les résultats de la section précédente montre que l'utilisation d'un système d'équilibrage dynamique pour le programme *press* ne donne pas des avantages qui justifient son utilisation. Cela est sûrement dû en grande partie à la nature peu dynamique, donc équilibré, de l'application.

Introduisons donc un déséquilibre artificiel. Comme notre architecture visée est un réseau de stations de travail Linux, nous pouvons imaginer qu'il existe encore d'autres applications non AMPI qui s'exécutent simultanément avec notre programme *press*. Modélisons cette charge extérieure non nécessairement équilibrée, par un processus qui occupe 66% de temps de calcul d'un seul processeur. Les autres processeurs proposent tout leur temps de calcul au programme test.

Pour simplifier les tests et leurs interprétations modifions les paramètres des tests. En effet, l'équipe de développement de CHARM++/AMPI propose aussi un logiciel de visualisation d'une exécution d'un programme AMPI, appelé *projections* [45]. A partir d'une trace enregistrée pendant l'exécution *projections* propose différents graphiques. Dans les tests de la section précédente ces traces étaient d'une telle grandeur (10 Mb) que *projections* n'arrivait pas à les ouvrir. Nous nous limitons donc à 10 itérations en tout, ce qui suffit pour des mesures de temps des itérations. En effet, vue la nature statique de l'application, les temps d'itération sont assez constants dans le temps. Laissons commencer le calcul du tenseur de pression à la 6^e étape. En augmentant le nombre d'atomes participant à la simulation on devait voir aussi des gains dus à la virtualisation des processeurs de AMPI par rapport à MPI. Prenons donc 15 * 15 atomes pour le solide et 15 * 15 * 15 atomes pour la monocouche. Les autres paramètres sont identiques au jeu de test 8.

Les tests de migration sont effectués à chaque itération et commencent pour des raisons de conception à partir de l'itération 1. Aussi ils ont été répétés avec différentes stratégies d'équilibrage dynamique de charge, à savoir une stratégie vide (AMPI), une stratégie spéciale pour des réseaux de stations de travail (AMPI WS) et une stratégie d'équilibrage défensive (AMPI Refine (Ref)).

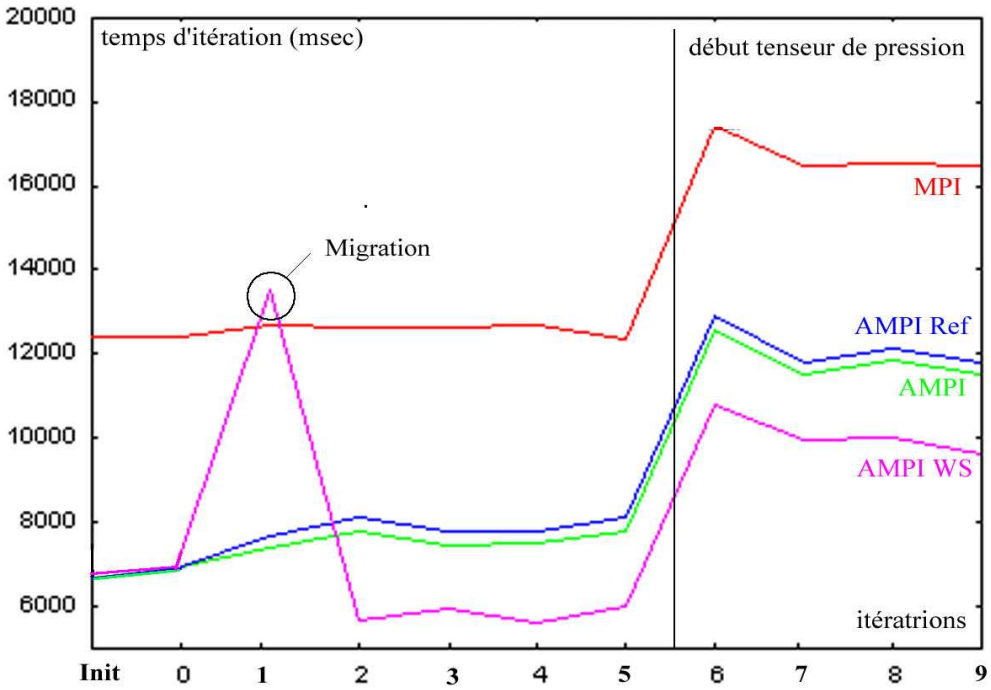


FIG. 5.18 – *Comparaison des temps d'itération avec une charge extérieure de la version* MPI *et de la version* AMPI *avec différentes stratégies, pour le jeu de test 13*

Les courbes des temps d'itération demandent une petite explication. D'abord nous remarquons que les versions AMPI avec les différentes stratégies d'équilibrage dynamique de charges, sont plus performantes en temps de calcul. Cela est dû au concept de virtualisation des processeurs.
L'effet de renversement de cette situation après le début du calcul du tenseur de pression comme dans la section précédente, ne se présente plus. La cause est le processeur chargé et le fait que les stratégies ne migrent pas des processeurs virtuels vers ce dernier. En effet le processeur chargé occupe tellement de temps que les pertes de temps dues à des fausses migrations vers des autres processeurs sont comblées (voir aussi figure 5.19 itération 8).

Comme pour l'application test, dans le cas de deux esclaves pour MPI, nous devons être conscient qu'nous testons dans cette expérience deux facteurs, à savoir la virtualisation et l'effet des stratégies de migration. Ainsi, la différence des temps d'itération entre MPI et AMPI sont due à la virtualisation. Si on fait abstraction de ce gain de virtualisation, la courbe des temps d'itération de MPI se situait probablement un peu en-dessous de la courbe AMPI sans stratégie[16]. Néanmoins, dans notre expérience nous pouvons aussi tester l'effet de migration en comparant entre-eux les versions AMPI avec différentes stratégies.
Ainsi, la stratégie "AMPI WS" nécessite une attention spéciale. Rappelons que les tests

[16]Comme c'est le cas après le début du tenseur de pression dans la section précédente.

de migration commencent à l'itération 1. La stratégie "WS" remarque que le processeur *P1* est chargé et migre un processeur virtuel de *P1* vers un autre processeur réel. Comme la migration exige la participation de *P1* chargé, cette étape de migration a un coût de temps élevé. Cela explique la hausse de la courbe d'"AMPI WS" en cette itération. Dans les itérations suivantes cette stratégie montre alors des gains de temps par rapport aux autres stratégies qui n'ont pas fait de migration à l'itération 1.
A l'itération 7, les deux stratégies réelles d'équilibrage dynamique des charges montrent encore les migrations dues à la fausse interprétation due à la nature de l'application. Comme déjà mentionné ci-dessus ces fausses migrations n'affectent pas, à cause de la charge extérieure, le temps d'une itération. Aussi le temps de la migration, où *P1* ne participe plus, est négligeable par rapport au temps que *P1* calcule. Ainsi, à l'inverse de la section précédente, les durées de tests de migration[17] ne se remarquent plus.
Comme on pouvait s'attendre la stratégie "AMPI Ref" ne détecte pas la possibilité des gains de temps avec une migration à partir du processeur chargé. En effet, elle ne prend pas en considération la charge extérieure. Néanmoins, elle permet de voir, qu'il ne faut pas migrer un processeur virtuel vers *P1* à l'itération 7.
Remarquons encore que les auteurs n'ont pas trouvés une explication satisfaisante pour les petites hausses des temps d'itération en plus du calcul du tenseur, qui se présente pour chaque version.

La figure 5.19 confirme les déclarations des paragraphes précédents. La figure montre l'exécution de la version AMPI avec les trois stratégies investiguées. Une ligne de temps *PE* représente l'activité d'un processeur réel. Les lignes verticales en rose séparent graphiquement les itérations. Par exemple, l'itération *Init* correspond à l'étape des conditions initiales. Les rectangles rouges sur les lignes des processeurs réel montrent l'activité d'un processeur virtuel. Sur *PE1* on remarque que les temps d'activités sont prolongés à cause de la charge extérieure.
Chaque processeur virtuel est identifié par son numéro. Ainsi on peut suivre une migration d'un processeur virtuel. Par exemple, dans le schéma de "AMPI WS" la migration du processeur virtuel 4 entre l'itération 1 et 2. Les rectangles rouges marqués avec un T à partir de l'itération 6, visualisent le temps de calcul du tenseur de pression.
Les rectangles blancs montrent des temps d'attente[18] du processeur réel. Les pourcentages d'occupation des processeurs sont alors montrés à gauche. En comparant ces pourcentages sur les trois schémas, ils montrent que la stratégie "AMPI WS" équilibre au mieux les charges. Spécialement *PE1* a donner la charge de calcul aux autres processeurs.
Nous voyons aussi la diminution des temps d'une itération après la migration de la stratégie "WS" après l'étape 1 par rapport aux deux autres stratégies. En effet, en comparant par exemple les longueurs de l'itération 2[19], c'est la stratégie 'WS' qui gagne. Mais comme déjà cité, la migration a provoqué un sur-coût pour "WS" à l'itération 1, due au fait qu'on migre à partir du processeur *PE1* chargé.
Le début du calcul du programme pour les stratégies d'équilibrage dynamique de charges semble être précédé par d'une période de lancement et initialisation de stratégie. Cette période est aussi sûrement prolongée par le processeur *PE1* chargé, et affecte le temps global de notre petit test, tel que la version avec la stratégie "AMPI WS" dure plus longtemps que avec une stratégie vide (AMPI). En augmentant la durée du test on peut facilement, comme les temps d'itérations réels le montrent, combler ce sur-coût. Aussi la période de lancement de stratégie affecte légèrement les pourcentages d'occupation des processeurs.

[17] encore remarquable par des petites barres dans la figure 5.16
[18] Pas d'activité d'un processeur virtuel AMPI.
[19] De même pour les itérations 3 à 9.

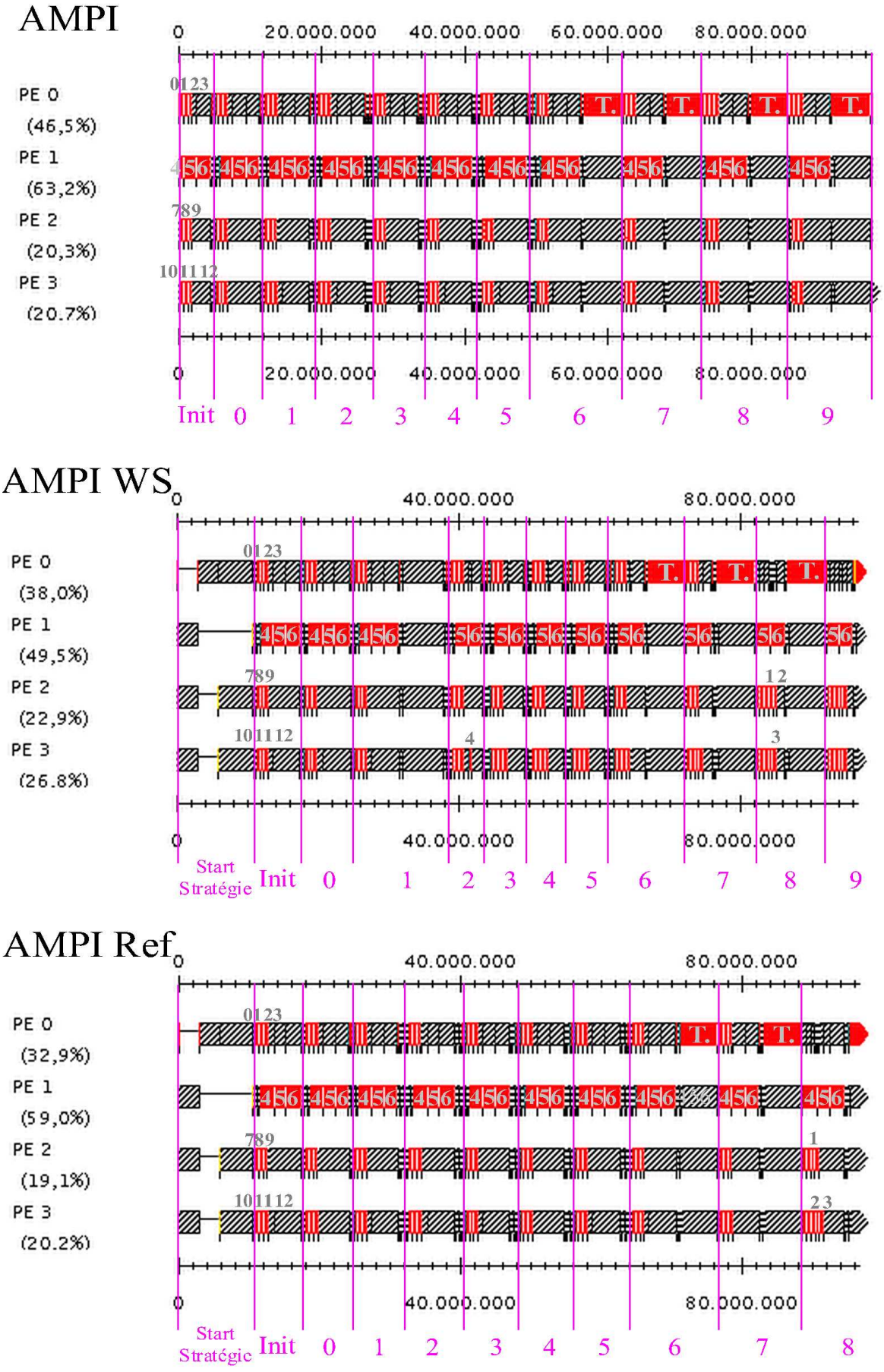


FIG. 5.19 – *Lignes de temps des processeurs avec une charge extérieure pour P1, pour le jeu de test 13*(Graphiques produits avec le logiciel *projetions*).

Finalement, remarquons encore que la stratégie "AMPI WS" pouvait faire mieux en migrant encore un processeur virtuel (numéro 5 ou 6) à partir de *PE1*.
Avant le calcul du tenseur de pression, une migration n'est pas envisagée car la stratégie est basée sur des mesures et sur l'hypothèse de persistance[20]. Ainsi, une migration de processeur virtuel 5 ou 6 augmentait le temps de calcul du processeur destinataire de telle manière que le temps global d'une itération augmentait. Or l'hypothèse de persistance s'avère fausse dans les conditions d'augmentation des temps de calcul due au charge extérieure. En effet, un processus virtuel de *PE1* s'exécutait plus rapidement sur un processeur sans charges, et le temps global d'une itération diminuait.
Après le début du tenseur la stratégie ne détecte pas la possibilité de migration car elle ignore comme pour les fausses migrations la nature réelle de l'application. En effet, elle essaye de minimiser le temps de *PE0* qui est maintenant le plus long, mais sans détecter qu'en migrant à partir de *PE1* le temps de *PE0* diminuait.
Notons encore une fois, que pour ce cas il est possible, en AMPI, de décrire sa propre stratégie qui est mieux adapté à la nature de synchronisation de l'application. Néanmoins, pour le programme *press* AMPI, la stratégie "WS" a montré de l'utilité dans le cas d'un déséquilibre artificiel créé avec une charge extérieure. En général, nous pouvons affirmer pour notre cas que "AMPI WS" est la plus performante des trois stratégies testées.

5.5.4 Remarques sur les passage d'MPI à AMPI

Un objectif important de ce chapitre était de voir les défis et difficultés d'un passage d'un code réel en MPI à un code en AMPI.

Comme pour notre programme test *sum* la conversion en AMPI demande quelques petits changements (voir aussi page 63). D'abord, nous devons privatiser les variables globales dans une structure, afin de pouvoir placer plusieurs *user-level threads* sur un même processeur. Aussi, nous spécifions la fin d'une itération comme le moment de l'exécution où une migration peu avoir lieu. Afin, de limiter le sur-coût de la gestion des migrations[21], notre programme *press* fait seulement tous les milles itérations un appel de test de migration. Pour la migration des variables allouées dynamiquement par un *thread* les fonctions *pup* ont été crées.

Outre ces changements standard, un autre aspect a posé problème pour la version AMPI du programme *press* est l'instabilité des versions de l'environnement AMPI. En effet, l'équipe de développement du *Parallel Programming Laboratory* de l'*University of Illinois at Urbana-Champaign* ne pratique pas une gestion de version pour l'environnement AMPI. Presque chaque jour il y a une version modifiée qui est seulement validée par un test automatique partiel. La dernière "version" a donc toutes les dernières modifications, mais n'est pas nécessairement stable.
Ainsi, nous avions eu deux versions différentes qui disposaient chaque fois deux caractéristiques intéressantes. Mais l'une était buggée dans une version et l'autre dans l'autre version. Avec l'aide de l'équipe de développement d'AMPI, nous sommes arrivés à combiner les deux versions pour avoir une version mixte qui possède toutes les caractéristiques désirées.

Un autre désavantage de cette évolution continuelle est que les documentations ne sont plus actuelles aux versions publiées ce qui complique l'utilisation et la compréhension du fonctionnement de l'environnement. Par exemple, la syntaxe de la fonction *pup* n'est plus

[20] le temps actuel est plus ou moins celle du futur

[21] Contrairement au programme *sum*, les coûts aux appels de test de migration sont non négligeables vu le faible temps d'exécution d'une itération (voire barre sur le graphique 5.16).

la même que dans le manuel d'AMPI.
La stratégie "AMPI WS" montrait aussi parfois des instabilité dans l'exécution. Ces erreurs n'étaient cependant pas périodiques ! ?
Ainsi, l'utilisation du logiciel de visualisation *projections* développé par le même laboratoire était parfois un peu limité.

Une différence surprenante entre l'implémentation Mpich du standard MPI et AMPI, qui est en général rien d'autre qu'une autre implémentation du standard MPI, a été constaté pour une fonction du standard. Il s'agit de la fonction `MPI_Type_vector` qui crée un type de donnée utilisateur qui est transférable par message MPI. En AMPI l'instruction `MPI_Type_vector(1, 4, 0, MPI_DOUBLE, &MPI_COORDNUM);` à le même effet que `MPI_Type_vector(1, 4, 4, MPI_DOUBLE, &MPI_COORDNUM);` en Mpich. Pire encore, en remplacant le troisième paramètre par 4 dans l'instruction pour AMPI le type résultat `MPI_COORDNUM` est différent qu'en Mpich. Une comparaison avec les spécification du standard MPI 1.1 [35] montre que l'implémentation Mpich est la seule correcte.

Une dernière difficulté, était due à la conception du programme *press*. La migration d'AMPI est limitée pour les fichiers. Il n'est pas possible de migrer les fichiers avec leur processeur virtuel. Or notre programme enregistre des valeurs instantanées à chaque itération. On doit fermer le fichier avant la migration pour le réouvrir après. Cela a comme désavantage qu'on a sur chaque processeur réel une partie du fichier total. Mais le seul processeur qui écrit des fichiers est le processeur virtuel "maître". Ainsi nous avons modifié le code des stratégies, telle que le processeur virtuel 0, donc le processeur virtuel "maître", était exclu de la migration.

Si on considère que l'environnement AMPI se stabilise avec le temps, nous pouvons conclure que la conversion d'un programme MPI en AMPI est faisable avec un effort limité. En attendant, l'expérience du test à montrer que, l'équipe de développement d'AMPI propose son aide volontairement pour éliminer des problèmes éventuels.

5.6 Conclusion à propos des tests

Pour ce qui concerne la parallélisation, on peut remarquer qu'on est arrivé à un programme qui montre des résultats dont l'ordre de grandeur est comparable à celle de la version séquentielle initiale. Le speedup est améliorable, mais peut être expliqué par le fait que le programme était parallélisé seulement à 75%, ce qui impose une limite théorique de 4 pour le speedup, et par la lenteur de réseau de la plateforme de test, un Ethernet 100 MB/s.
Une autre perspective de ce travail de parallélisation est une conception plus générique du programme parallèle, afin de rendre le programme plus portable est plus performante. Aussi, une adaptation du programme *press* à la dernière version de la nature physique de la simulation[22] est nécessaire pour pouvoir le comparer avec le programme séquentiel finale qui calcule la simulation.

La transformation de MPI en AMPI est au moins théoriquement un effort limité.
La majeur critique négative pour AMPI est que l'environnement est encore en plein développement et est dés lors instable. Remarquons aussi l'absence des gains d'un équilibrage dynamique pour le programme *press* sans déséquilibre externe. Cela est dû à la nature

[22] Au moment du développement du programme *press* certaines facteurs de la simulation n'était pas encore spécifiés.

trop statique de l'application. Il faudrait répéter l'expérience avec la deuxième partie de la simulation qui est plus dynamique.
Pour ce qui est de la charge extérieure, AMPI avec la stratégie "WS" montre clairement ces avantages par rapport à MPI. En plus le résultat est encore améliorable par la possibilité de développement d'une stratégie adaptée à la nature de synchronisation de l'application.

Conclusions

À partir d'un cadre de calcul parallèle et à travers des concepts et principes théoriques des supports d'exécution avec équilibrage dynamique de charge, ce travail avait pour objectif de faire une évaluation théorique des supports du domaine. Suite à cette évaluation, l'environnement AMPI a été choisi, dans le cadre de ce mémoire, pour des tests pratiques. En effet, avec ses capacités d'équilibrage dynamique de charge et la possibilité de réutilisation de code MPI, le système AMPI était le mieux adapté aux contraintes posées.

L'objectif pour les tests pratiques était de valider les optimisations effectuées par le système AMPI. Dès qu'un facteur dynamique se présente, les différents résultats des tests montrent que le système d'équilibrage de charge d'AMPI a des comportements intéressants et efficaces. Ainsi, notre validation d'AMPI est positive pour des applications irrégulières et dynamiques dans un contexte hétérogène comme un cluster avec charge extérieure.
Les résultats de l'application pratique du dernier chapitre parfois moins favorables pour AMPI sont au moins en partie explicables par la nature statique de la simulation calculée. Une perspective de ce travail est d'envisager des tests avec des applications d'une nature plus dynamique. Ceci est par exemple le cas pour la deuxième partie de la simulation présentée qui concerne l'étalement de la monocouche.
En général, les résultats obtenus dans nos tests sont encore améliorables, surtout au niveau du concept de virtualisation ; les applications investiguées dans ce document n'ont pas pu montré davantage d'efficacité par rapport à un environnement classique MPI. Au niveau d'équilibrage, une situation sous-optimale est souvent atteinte lors de l'exécution des stratégies de migration. Aussi, l'immaturité de ce jeune environnement en développement qui a débouché entre autres sur de multiples problèmes d'instabilité au travers des tests pratiques est un élément négatif.
Le nombre limité de tests présentés dans le cadre de ce mémoire est à compléter pour une validation plus fine de l'environnement AMPI. Néanmoins, les tests préliminaires de ce document donnent une première évaluation. Nous pouvons donc dire que notre objectif de validation est atteint et les résultats de ce mémoire et la grande activité[23] de recherche du laboratoire origine d'AMPInous incitent à suivre l'évolution future de Charm++ et ainsi d'AMPI. En effet, ce dernier permet d'appliquer un mécanisme d'équilibrage dynamique de charge sur des applications MPI ordinaires de type dynamique, avec des efforts limités. Notre mémoire montre néanmoins que le concept (à la fois les principes et la technologie) d'équilibrage dynamique de charge est loin d'être mature et nécessite encore des recherches approfondies. L'avènement des *Grids* souligne encore plus l'importance de cet effort.

Une conclusion complémentaire dépassant ce travail est à la fois d'une nature scientifique et personnelle.
Elle se base sur l'implémentation des concepts théoriques dans la pratique. Ainsi, la transposition des idées issues de notre parcours de l'état de l'art dans un cadre pratique ne se passe pas sans difficultés ni surprises. Par exemple, le concept de virtualisation, très

[23] Cette activité de recherche se montrait entre-autre dans plusieurs contacts avec le laboratoire pour la mise en place de l'environnement AMPI pour les tests pratiques.

prometteur dans la littérature, ne pouvait pas montrer ses gains de performances dans les tests réels.
Même avec notre simple application test ou pour l'instabilité numérique du programme *press*, il était difficile de contrôler toutes les variables qui ont une influence sur le résultat afin d'avoir des mesures représentatives pour valider les idées théoriques. Dans une approche expérimentale, comme celle suivie dans ce mémoire, un regard critique est donc à mettre sur l'interprétation des résultats en situant ces mesures dans leurs contextes relatifs. Comme nous l'avons fait dans notre démarche, en se basant sur notre formation scientifique, nous devrons analyser de tels résultats avec un maximum d'objectivité.

Bibliographie

[1] Adaptative MPI Manual, *http://charm.cs.uiuc.edu/manuals/pdf/ampi.pdf* (Last revised 07/08/03) (Date of access 18/09/03).

[2] Athapascan-0 : A Parallel Programming Environment, *http://www-id.imag.fr/Logiciels/ath0/*, (Last revised 08/01/03) (Date of access 26/03/04).

[3] Bertrand E., *Mémoire : Dynamique d'étalement de monocouches par dynamique moléculaire*, Université Mons-Hainaut, 2004.

[4] Bisseling R.H., *Parallel Scientific Computation : A Structured Approach using BSP and MPI*, Oxford University Press, 2004.

[5] BLAS (Basic Linear Algebra Subprograms), *http://www.netlib.org/blas/*, (Date of access 23/04/04).

[6] Blumofe R. D. et Papadopoulus D., *The perfomance of Work Stealing in Multiprogrammes Environnements*, University of Texas at Austin, 1998.

[7] Born M. et von Karmann Th., *Über Schwingungen in Raumgittern*, Physikalische Zeitschrift, Num. 13, page 297-309, 1912.

[8] Butenhof D.R., *Programming with Posix Threads*, Addison-Wesley, 1997.

[9] Buyya R., *High Performance Cluster Computing (Vol 1 : Architecture and Systems)*, Prentice Hall PTR, New Jersey, 1999.

[10] Buyya R., *High Performance Cluster Computing (Vol 2 : Programming and Application)*, Prentice Hall PTR, New Jersey, 1999.

[11] Chao Huang, Orion Lawlor, L. V. Kalé, *Adaptive MPI*, Parallel Programming Laboratory, Univerity of Illinois at Urban-Champaign, 2003.

[12] CHARM++, *http://charm.cs.uiuc.edu* (Last revised 07/08/03) (Date of access 18/09/03).

[13] CHARM++ Frequently Asked Questions, *http://charm.cs.uiuc.edu/faq.shtml* (Last revised 31/05/02) (Date of access 17/09/03).

[14] The CHARM++ Programming Language Manual, *htt://charm.cs.uiuc.edu/manuals/pdf/charm++.pdf* (Date of access 18/09/03).

[15] Chaussumier F., Desprez F. et Prylli L., *Rapport de recherche : Asynchronous Communication in MPI - the BIP/Myrinet Approach*, Institut National de Recherche en Informatique et en Automatique, France, 2000.

[16] Flynn M. J.,*Some Computer Organizations and their Effectiveness*, IEEE Transactions on Computers, Vol. C-21, 1972.

[17] The Cilk Project, *http://supertech.lcs.mit.edu/cilk*, (Last revised 16/12/02) (Date of access 26/03/04).

[18] Friedrich R. et Ortmann M., *Load Balancing : Seminar Datenverarbeitung*, Seminare Datenverarbeitung Wintersemester 2000/2001, Lehrstuhl für Datenverarbeitung, Ruhr-Universität Bochum, 2001.

[19] Gear C.W., *The Automatic Integration of Ordinary Differential Equations*, Prentice-Hall, New Jersey, 1971.

[20] Gengler M., Ubéda S. et Desprez F., *Initiation au Parallélisme : Concepts, Architectures et Algorithme*, Masson, France, 1995.

[21] Genter F., *Thèse : Mouillage forcé et spontané à l'échelle nanométrique par dynamique moléculaire*, Université de Mons-Hainaut, 2004.

[22] Harchol-Balter M. et Downey A., *Exploiting Process Lifetime Distributions for Dynamic Load Balancing*, ACM Transactions on Computer Systems, Vol. 15, Num. 3, pages 253-285, 1997.

[23] High Perfomance Fortran (HPF), *http://www.crpc.rice.edu/HPFF* (Date of access 15/03/04).

[24] Overview Parallel Architecture, *http://www.top500.org/ORSC/2003/overview.html*, (Last revised 28/10/03) (Date of access 26/03/04).

[25] Horstmann C. S. et Cornell G., *Au coeur de Java2 : Fonctions avancées*, Campus Press, Paris, 2000.

[26] HPC Hardware Technology, *http://www.epcc.ed.ac.uk/HPCinfo/hardware.html* (Date of access 31/10/03).

[27] Java : The sources for Developers, *http://java.sun.com*, (Date of access 11/04/04).

[28] Kalé V. L., *The virtualization model of parallel programming : Runtime optimizations and the state of art*, LACSI 2002, Albuquerque, 2002.

[29] Kalé L. V. Ramkumar B. Sinha A. B. Gürsoy A., *The Charm Parallel Programming Language and System : Part I - Description of Language Features*, Technical Report 95–2, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana-Champaign, 1995.

[30] Kumar V., Grama A., Gupta A. et Karypis G., *Introduction to Parallel Computing (Design and Analysis of Algorithmes)*, Benjamin Cummings, California, 1994.

[31] LAM / MPI Parallel Computing, *http://www.lam-mpi.org*, (Last revised 14/03/04) (Date of access 11/04/04).

[32] Leiserson C. E. et Prokop Harald, *A Minicourse on Mulitthreaded Programming*, MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1998.

[33] Manneback P., *Projet de recherche : Maximalisation de l'Efficacité des Clusters de Calcul Avec un Nouvel Ordonnaceur (MECCANO)*, Service d'Informatique, Faculté Polytechnique de Mons, 2003.

[34] Mounié G., *Thèse : Ordonnancement efficace d'application parallèles : les tâches malléables monotones*, Institut National Polytchnique de Grenoble, France, page 10-16, 2000.

[35] The Message Passing Interface (MPI) standard, *http://www-unix.mcs.anl.gov/mpi/* (Date of access 06/10/03).

[36] MPICH - A portable implementation of MPI, *http://www-unix.mcs.anl.gov/mpi/mpich*, (Date of access 30/08/03).

[37] Namyst Raymond, *Contribution à la conception de supports exécutifs multithreads performants*, Habiliation à diriger les recherches, École normale supérieur de Lyon, 2001.

[38] Needham S. et Hansen T., *Cluster Programming Environments*, School of Computer Science and Software Engineering Monash University, Melbourne, Australia, 1999.

[39] Nexus, *http://www.globus.org/nexus*, (Date of access 12/04/04).

[40] OpenMP : Simple, portable, scalable SMP Programming, *http://www.openmp.org*, (Date of access 26/03/04).

[41] Pacheco P. S., *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, 1997.

[42] PVM - Parallel Virtual Machine, *http://www.csm.ornl.gov/pvm/pvm_home.html* (Last revised 09/03/04) (Date of access 15/03/04).

[43] Parallel Programming Environments, *http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/ProgEnvs.htm* (Last revised 14/08/99) (Date of access 06/10/03).

[44] Parallel Systems - Architecture : Classifications, *http://www.cems.uwe.ac.uk/teaching/notes/parallel/architec/arch100.htm* (Last revised 08/01/97) (Date of access 03/11/03).

[45] Projections Manual, *http://charm.cs.uiuc.edu/manuals/html/projections/manual.html* (Last revised 12/12/2003) (Date of access 15/12/2003).

[46] Roch J.-L., Gautier T. et Revire R., *Athapascan : an API for Asynchronous Parallel Programming - User's Guide*, Rapport technique à l'INRIA, Num. 0276, France, 2003.

[47] Rovillard S., *La dynamique moleculaire*, Chapitre 2, *http://crmm.umh.ac.be/Files/Courses/ModMol/cours/modmol.pdf*.

[48] The ScaLAPACK Project *http://www.netlib.org/scalapack/*, (Date of access 23/04/04).

[49] Shahzad M., *Dynamic Load Balancing in a Network of Workstations*, 95.515F Research Report, Carleton University, 2000.

[50] Critical : High-performance Computing (WMPI), *http://www.criticalsoftware.com/hpc*, (Last revised 03/03/04) (Date of access 11/04/04).

Annexe A

Tableaux de tests de l'application pratique

A.1 Jeux de tests

Jeux de tests	**test 1**	**test 2**	**test 3**	**test 4**	**test 5**	**test 6**	**test 7**	**test 8**	**test 13**
nbrAtomeCoteSol	3	4	5	6	7	8	9	10	15
distAtomeSol	1	1	1	1	1	1	1	1	1
nbrAtomeChaîneMem	3	4	5	6	7	8	9	10	15
nbrChaîneCoteMem	3	4	5	6	7	8	9	10	15
tempRef	300	300	300	300	300	300	300	300	300
rayonCoupure	5	8	10	12	15	15	15	15	15
rapSol	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000
rapMem	5	5	5	5	5	5	5	5	5
rapBaseChaîne	100	100	100	100	100	100	100	100	100
nbrIterEquilibre	30 000	30 000	30 000	30 000	30 000	30 000	30 000	30 000	10
nbrIterScaling	10 000	10 000	10 000	10 000	10 000	10 000	10 000	10 000	3
nbrIterAttente	10 000	10 000	10 000	10 000	10 000	10 000	10 000	10 000	3
initRand	14 789	1 234	12	12	12	12	12	12	12

A.2 Temps d'exécution des différentes étapes d'une itération

(msec)*	Programme C séquentiel d'Emilie						
	tps Iter	tps Pred	tps Eval	tps Cor	tps Haut	tps Tens	tps Temp
test1	2	0	1	0	0	0	0
test2	18	0	14	0	0	3	0
test3	69	0	53	0	0	14	0
test4	190	1	147	0	0	41	0
test5	559	2	432	0	0	123	0
test6	908	3	693	1	0	210	0
test7	1383	5	1044	1	0	331	0
test8	1990	7	1485	1	0	495	0
	Programme C structuré séquentiel						
	tps Iter	tps Pred	tps Eval	tps Cor	tps Haut	tps Tens	tps Temp
test1	4	0	2	0	0	0	0
test2	22	1	15	1	0	4	0
test3	79	3	56	2	0	15	0
test4	214	5	160	3	0	44	0
test5	594	8	448	6	0	130	0
test6	966	12	719	8	0	225	0
test7	1425	18	1043	12	0	351	0
test8	2078	24	1498	17	0	538	0

* mesures effectuées sur un Intel Pentium 4 (2.00 Ghz, 256 Mo Ram)

Annexe B

Explications des formules physiques

Deuxième loi de Newton ou l'équation classique du mouvement

La deuxième loi de Newton s'énonce comme suite : "Le changement de mouvement est proportionnel à la force motrice imprimée, et s'effectue suivant la droite par laquelle cette force est imprimée".

Une force dans le langage de Newton, c'est donc ce qui provoque le "changement du mouvement". Pour Newton, "La quantité d'un mouvement est la mesure que l'on tire à la fois de sa vitesse et de sa quantité de matière", autrement dit le produit de sa masse par sa vitesse.

$$\vec{p} = m\vec{v}$$

En utilisant les symboles mathématiques modernes, la première partie de cette deuxième loi peut alors se reformuler.
La force $\vec{f}$ est égale à la variation en fonction du temps de la quantité de mouvement, soit :

$$\vec{f} = \frac{d\vec{p}}{dt} = \frac{dm\vec{v}}{dt}$$

Si la vitesse est significativement inférieure à celle de la lumière, nous pouvons supposer que la masse ne varie pas en fonction de la vitesse[1]. Ainsi nous trouvons l'équation classique du mouvement :

$$\vec{f} = m\frac{d\vec{v}}{dt} = m\vec{a}$$

Travail

Si un point de masse m subit un déplacement élémentaire $d\vec{r}$ sous l'effet d'une force $\vec{f}$, cette force effectue un travail élémentaire valant par définition :

$$dW = \vec{f}\, d\vec{r}$$

Si cette masse m est déplacée d'un endroit A à un endroit B, le travail total est :

$$W_{A,B} = \int_A^B \vec{f}\, d\vec{r}$$

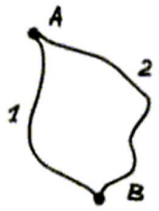
[1] En dynamique moléculaire les vitesses sont importantes pour des dimensions très réduites au niveau des atomes et molécules. Cependant, ces aspects de la relativité et de la mécanique quantique sont ignorés étant donné que la dynamique moléculaire est basée sur des données empiriques englobant implicitement tous les effets relativistes et quantiques.

Énergie potentielle

Si le travail de la force $\vec{f}$ entre les points A et B ne dépend pas du chemin suivi, on dit que cette force dérive d'une énergie potentielle ou bien que le champ de force est "conservatif" (contre-exemple : dans un mouvement avec frottement le travail dépend nécessairement de la voie choisie).
Soit alors deux points A et B de l'espace. Il y a plusieurs chemins possibles pour joindre ces deux points. Si l'on en choisit deux au hasard et que la champ est conservatif on a :

sur le 1^{er} chemin : $\int_A^B \vec{f}\, d\vec{r}_1$

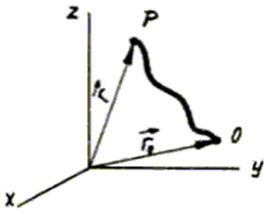
sur le $2^{ème}$ chemin : $\int_A^B \vec{f}\, d\vec{r}_2$



Si le champ est conservatif on a : $\int_A^B \vec{f}\, d\vec{r}_1 = \int_A^B \vec{f}\, d\vec{r}_2$

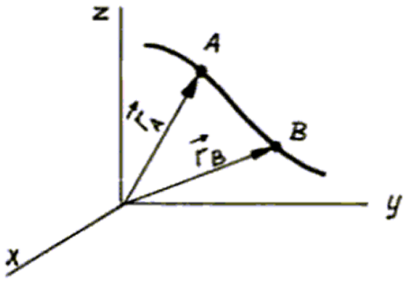
Le travail en jeu est donc une fonction du lieu seul (U) c'est-à-dire dépendant uniquement du point de départ et du point d'arrivée. En effet, si le travail dépendait du chemin, il serait possible de choisir la voie la plus généreuse quand le système fournit du travail et la voie la plus économique quand on la ramène à l'état initial. Ce serait donc un mouvement perpétuel et le principe de conservation de l'énergie l'interdit.
Attachons alors à chaque point P du champ de force une valeur de la fonction (un nombre réel) correspondant au travail effectué par le champ de force lorsque le mobile passe d'un point P à 0, 0 étant un point de référence choisi arbitrairement. Donc par définition :

$$U(P) = \int_P^0 \vec{f}\, d\vec{r} \qquad \text{avec } U(0) = 0$$



En généralisant cette définition, on dira que le travail effectué par une force conservative, lorsque le mobile passe de A à B, est égal à la diminution d'énergie potentielle entre A et B :

$$W_{A,B} = \int_A^B \vec{f}\, d\vec{r} = -(U(B) - U(A))$$



L'équation précédente s'utilise très souvent sous forme différentielle soit :

$$dW = - dU$$

ou encore

$$\vec{f}\, d\vec{r} = - dU$$

Annexe C

Exemple pour une erreur d'arrondi sur ordinateur

Sur l'ordinateur l'addition et la soustraction de nombres réels dont le résultats est d'un ordre de grandeur très différent provoque des erreurs d'arrondi. En effet, l'ordinateur stocke seulement un certain nombre de chiffres significatifs pour la représentation des nombres réels.
Montrons cela à l'aide d'un petit programme d'exemple. Supposons qu'on veuille additionner des nombres presque opposés comme dans le programme *press*. Ainsi leurs sommes est d'un ordre de grandeur plus petit. Simplifions le problème et travaillons avec deux nombres qui soient alors "$cos(0.1) * 10^{20}$" et "$-cos(0.1 + 0.0001) * 10^{20}$"[1]. Exécuter le programme suivant, qui en mathématique devait donné le même résultat pour $res1$ et $res2$:

$double\ res1, res2;$
$res1 = 0.;$
$res2 = 0.;$

$res1 = res1 + (cos(0.1) * pow(10, 20));$ $//(1)$
$res2 = res2 - (cos(0.1 + 0.0001) * pow(10, 20));$ $//(1)$

$res1 = res1 - (cos(0.1 + 0.0001) * pow(10, 20));$ $//(2)$
$res2 = res2 + (cos(0.1) * pow(10, 20));$ $//(2)$

Remarquons qu'on a simplement inversé l'ordre d'addition. Le résultat est mathématiquement surprenant :

$res1(1)$ $9950041652780258 0992.000000$
$res2(1)$ $-9949941769613569 2288.000000$
$res1(2)$ 998831666886344.000000
$res2(2)$ 998831666889768.000000

$res1 - res2 = -3424.000000$

Ainsi, on peut comprendre si de telles erreurs d'arrondissement se répète ou si on continue de faire des calcules avec $res1$ ou $res2$ que l'erreur peut grandir et donner lieu à des résultats différents.

[1] Remarquons, comme l'erreur se produit sur la mantisse que l'exemple ne dépend pas de l'exposant choisi.

Annexe D

Le code du programme sum

D.1 en MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "mpi.h"

int nbrProc;        /* nombre de processus */

double m;           /* nombre d'opération par itération par esclave */
double n;            /* nombre d'itération */

/*Affiche début du programm*/
void debut(){
  double S;

  printf("\n  Programm MPI Sum V1\n");
  printf("  ==================\n\n");
  printf("1 maitre dirige %d esclaves qui font %.0f itérations avec
         %.0f opérations d'addition de l'entier 1\n",nbrProc-1,n,m);
  S = m * n * (double)(nbrProc-1);
  printf("Donc le programm fait %.0f opérations d'addition\n\n",S);
  printf("Le calcul est lance ...\n");
} /*debut*/

/*Affiche fin du programm*/
void fin(){
  fflush(stdout);
  printf("Le Programm s'est terminé correctement\n");
} /*fin*/

/*Calcule la différence en msec entre deux structures timeval*/
long diffTimeval (struct timeval *tvdeb, struct timeval *tvfin)
{
   long val;
   val=tvfin->tv_sec-tvdeb->tv_sec;
   val*=1000000;
   val+=tvfin->tv_usec;
   val-=tvdeb->tv_usec;
   return (val/1000);
} /*diffTimeval*/

/*La partie du maître*/
void maitre(){
    double S = 0;  /*nbr opérations globale*/
    double s = 0;  /*nbr opérations d'une itération sur un esclave*/
```

```
    double ss = 0; /*nbr opérations d'une itération par tous les esclaves*/
    int commande;  /*numero de commande: 1 = faire une itération, 0 = fin de calcul*/

    struct timeval tempsAvant;  /*temps avant le calcul*/
    struct timeval tempsApres;  /*temps après le calcul*/
    long temps;                 /*temps du calcul*/

    struct timeval tempsAvantIter;  /*temps début itération*/
    struct timeval tempsApresIter;  /*temps fin une itération*/
    long tempsIter;                 /*temps d'itération*/

    double i;    /*variable temporaire*/

    FILE *tps;   /*fichier pour les temps d'itération*/

    tps = fopen("./temps.dat","wt");

    //prendre le temps avant
    gettimeofday(&tempsAvant, 0);

    //effectuer les opérations d'addition
    commande = 1;   // 1 = faire une itération
    for(i=0;i<=n-1;i++){
      //prendre temps début itération
      gettimeofday(&tempsAvantIter, 0);

      //envoyer à tous le monde: faire une itération
      MPI_Bcast(&commande, 1, MPI_INT, 0, MPI_COMM_WORLD);

      //reçevoir de tous le monde le nbr d'opérations d'addition effectué
      MPI_Reduce(&s, &ss, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

      //calculer le nbr globale d'opération déjà effectué
      S += ss;

      // prendre temps fin d'itération
      gettimeofday(&tempsApresIter, 0);
      // enregistrment de temps d'itéraiton
      tempsIter = diffTimeval(&tempsAvantIter, &tempsApresIter);
      fprintf(tps,"%.0f %d\n",i+1,tempsIter);

    }

    //prendre le temps aprés
    gettimeofday(&tempsApres, 0);
    //calculer temps
    temps = diffTimeval(&tempsAvant, &tempsApres);

    //afficher S et temps
    fflush(stdout);
    printf("Le nombre d'opération effectué : %.0f\n",S);
    printf("Temps de calcul: %d (msec) (+/- %d  min)\n",temps,(temps/1000)/60);

    //demander fin de calcul
    //envoyer à tous le monde: demande de fin de calcul
    commande = 0;          // 0 = demander fin de calcul
    MPI_Bcast(&commande, 1, MPI_INT, 0, MPI_COMM_WORLD);

    fclose(tps);
}

/*La partie de l'esclave*/
void esclave(int monRang){
```

```
    int charge;    /*charge permettant de simuler une certaine charge*/
    double s = 0;  /* nbr opérations d'une itération sur l'esclave*/
    double ss = 0; /* nbr opérations d'une itération par tous les esclaves*/
    int commande;  /* numero de commande: 1 = faire une itération, 0 = fin de calcul*/

    double i;
    int j;  /*variables temporaires*/

    int iter=0;   /*nbr d'itération*/

    //recevoir et analyser une commande du maître
    MPI_Bcast(&commande, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //itération(si commande == 1)
    while(commande){

      //initialiser la charge
      //CHARGE VOID ou EXT
      //charge = 1;
      //CHARGE RANG
      //charge = monRang;
      //CHARGE DYNAMIQUE
      if (iter<n/2){
        charge = monRang;
      } else {
        charge = (nbrProc-1)  - monRang;
      }

      //effectuer une itération (en simulant la charge)
      for (j=0; j<charge; j++){
        s = 0;
        //faire m opérations d'addition
        for (i=0; i<=m-1; i++) {
          s++;
        }
      }

      //envoyer le nbr d'opérations addition et faire la somme des nbrs d'opérations
      //des autres esclaves
      MPI_Reduce(&s, &ss, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

      //recevoir et analyser une commande du maître
      MPI_Bcast(&commande, 1, MPI_INT, 0, MPI_COMM_WORLD);

      iter++;
    }
}

/**/
int main(int argc, char** argv){

  int monRang;         /* rang du processus */

  char host[20];         /*le nom de la machine sur lequel un processus tourne*/

  // INITIALISATION DE L'ENVIRONNEMENT MPI
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &monRang);
  MPI_Comm_size(MPI_COMM_WORLD, &nbrProc);

  // VERIFICATION DES PARAMETRES
  if (!monRang){
    if (argc != 3){
```

```
      fprintf(stderr,"Utilisation testSumV<x> <Nombre d'opérations addition
              par itération par esclave> <Nombre d'itération> \n");
      MPI_Abort(MPI_COMM_WORLD, 1);
      return 0;
    }
  }

  // INITIALISATION DES PARAMETRES
  m = atof(argv[1]);
  n = atof(argv[2]);

  // DEBUT DU PROGRAMM
  if (!monRang)
    debut();

  // PARTIE MAITRE
  if (!monRang){
    maitre();
  }

  // PARTIE ESCLAVE
  else{
    esclave(monRang);
  }

  // FERMETURE DE L'ENVIRONNEMENT MPI
  gethostname(host, 20);
  printf("fin processus %d sur %s\n",monRang,host);
  MPI_Finalize();

  // FIN DU PROGRAMM
  if (!monRang)
     fin();
  return 0;

} /*main*/
```

D.2 en AMPI

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#include "ampi.h"

/*AMPI variables globales*/
struct shareddata{
  int nbrProc;      /* nombre de processus */
  double m;         /* nombre d'opération par itération par esclave */
  double n;         /* nombre d'itération */
  int periodLB;     /* nombre d'itération entre deux tests de migration*/
};

/*AMPI pack-unpack sous-routine*/
void shareddataPup(pup_er p, struct shareddata **cpc){
  struct shareddata *c;

 //allocation de mémoire chez le nouveau processeur
  if (pup_isUnpacking(p)){
    *cpc = (struct shareddata *)malloc(sizeof(struct shareddata));
  }
```

```
  //sérialisation et désérialisation
  c = *cpc;
  pup_int(p,&c->nbrProc);
  pup_double(p,&c->m);
  pup_double(p,&c->n);
  pup_int(p,&c->periodLB);

  //libération de mémoire chez l'ancien processeur
  if(pup_isPacking(p)){
    free(c);
  }
}

/*Affiche début du programm*/
void debut(struct shareddata *c){
  double S;

  printf("\n  Programm AMPI Sum V2\n");
  printf("  ====================\n\n");
  printf("1 maitre dirige %d esclaves qui font %.0f itérations avec
         %.0f opérations d'addition de l'entier 1\n",c->nbrProc-1,c->n,c->m);
  S = c->m * c->n * (double)((c->nbrProc)-1);
  printf("Donc le programm fait %.0f opérations d'addition\n",S);
  printf("Test de migration tous les %.0f iterations\n\n",c->periodLB);
  printf("Le calcul est lance ...\n");
} /*debut*/

/*Affiche fin du programm*/
void fin(){
  fflush(stdout);
  printf("Le Programm s'est termine correctement\n");
} /*fin*/

/*Calcule la différence en msec entre deux structures timeval*/
long diffTimeval (struct timeval *tvdeb, struct timeval *tvfin)
{
   long val;
   val=tvfin->tv_sec-tvdeb->tv_sec;
   val*=1000000;
   val+=tvfin->tv_usec;
   val-=tvdeb->tv_usec;
   return (val/1000);
} /*diffTimeval*/

/*La partie du maître*/
void maitre(struct shareddata *c){
    double S = 0; /*nbr opérations globale*/
    double s = 0; /*nbr opérations d'une itération sur un esclave*/
    double ss = 0;/*nbr opérations d'une itération par tous les esclaves*/
    int commande; /*num de cmde: 1 = faire une itér, 2 = demande de migration, 0 = fin de calcul*/

    struct timeval tempsAvant;  /*temps avant le calcul*/
    struct timeval tempsApres;  /*temps après le calcul*/
    long temps;                 /*temps du calcul*/

    struct timeval tempsAvantIter;  /*temps début itération*/
    struct timeval tempsApresIter;  /*temps fin une itération*/
    long tempsIter;                 /*temps d'itération*/

    char hostAvant[20];  /*le nom de la machine du maître avant une migration*/
    char hostApres[20];  /*le nom de la machine du maître après une migration*/

    double i;  /*variable temporaire*/
```

```
FILE *tps; /*fichier pour les temps d'itération*/


tps = fopen("./temps.dat","wt");

//AMPI Enregistrer les données du chunk
MPI_Register(&c,(MPI_PupFn) shareddataPup);

//prendre le temps avant
gettimeofday(&tempsAvant, 0);

//effectuer les opérations d'addition
commande = 1;   // 1 = faire une itération
for(i=0;i<=c->n-1;i++){
  //prendre temps début itération
  gettimeofday(&tempsAvantIter, 0);

  //determiner si test de migration
  if (fmod(i,c->periodLB)==0.&& i != 0.){
    commande=2;   // 2 = demander test de migration
  }

  //envoyer à tous le monde: faire une itération
  MPI_Bcast(&commande, 1, MPI_INT, 0, MPI_COMM_WORLD);

  //AMPI migrer éventuellement
  if(commande==2){
    printf("test Migration à l'itération %.0f\n",i);
    gethostname(hostAvant, 20);
    MPI_Migrate();
    gethostname(hostApres, 20);
    if(strcmp(hostApres,hostAvant)!=0){
      printf("processus 0 a migré de %s a %s\n",hostAvant,hostApres);
    }
    /*else {
      printf("processus 0 sur %s n'a pas migre\n",hostAvant);
    }*/
    commande=1;
  }

  //reçevoir de tous le monde le nbr d'opérations d'addition effectué
  MPI_Reduce(&s, &ss, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  //calculer le nbr globale d'opération déjà effectué
  S += ss;

  // prendre temps fin d'itération
  gettimeofday(&tempsApresIter, 0);
  // enregistrment de temps d'itéraiton
  tempsIter = diffTimeval(&tempsAvantIter, &tempsApresIter);
  fprintf(tps,"%.0f %d\n",i+1,tempsIter);
}

//prendre le temps aprés
gettimeofday(&tempsApres, 0);
//calculer temps
temps = diffTimeval(&tempsAvant, &tempsApres);

//afficher S et temps
fflush(stdout);
printf("Le nombre d'opération effectué : %.0f\n",S);
printf("Temps de calcul: %d (msec) (+/- %d  min)\n",temps,(temps/1000)/60);
```

```
    //demander fin de calcul
    //envoyer à tous le monde: demande de fin de calcul
    commande = 0;   // 0 = demander fin de calcul
    MPI_Bcast(&commande, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //liberation de mémoire
    fclose(tps);
    free(c);
}

/*La partie de l'esclave*/
void esclave(struct shareddata *c, int monRang){
    int charge;    /* charge qui permettant de simuler une certaine charge*/
    double s;      /* nbr opérations d'une itération sur l'esclave*/
    double ss = 0; /* nbr opérations d'une itération par tous les esclaves*/
    int commande;  /* num de cmde: 1 = faire une itér, 2 = demande de migration,  0 = fin de calcul*/

    char hostAvant[20];    /*le nom de la machine de l'esclave avant migration*/
    char hostApres[20];    /*le nom de la machine de l'esclave après migration*/

    double i;              /*variable(s) temporaire(s)*/
    int j;

    int iter=0;            /*nbr d'itération*/

    //AMPI Enregistrer les données du chunk
    MPI_Register(&c,(MPI_PupFn) shareddataPup);

    //recevoir et analyser une commande du maître
    MPI_Bcast(&commande, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //itération(si commande == 1)
    while(commande){

      //initialiser la charge
      //CHARGE VOID ou EXT
      //charge = 1;
      //CHARGE RANG
      //charge = monRang;
      //CHARGE DYN
      if (iter<n/2){
        charge = monRang;
      } else {
        charge = (nbrProc-1)  - monRang;
      }

      //effectuer une itération (en simulant la charge)
      for (j=0; j<charge; j++){
        s = 0;
        //faire m opérations d'addition
        for (i=0; i<=m-1; i++) {
          s++;
        }
      }

      //envoyer le nbr d'opérations addition et faire la somme des nbrs d'opérations
      //des autres esclaves
      MPI_Reduce(&s, &ss, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

      //recevoir et analyser une commande du maître
      MPI_Bcast(&commande, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
      //AMPI migrer éventuellement
      if(commande==2){
        gethostname(hostAvant, 20);
        MPI_Migrate();
        gethostname(hostApres, 20);
        if(strcmp(hostApres,hostAvant)!=0){
          printf("processus %d a migre de %s a %s\n",monRang,hostAvant,hostApres);
        }
        /*else {
          printf("processus %d sur %s n'a pas migre\n",monRang,hostAvant);
        }*/
        commande=1;
      }

      iter++;
    }
}

/**/
int MPI_Main(int argc, char** argv){

  struct shareddata *c;   /*AMPI Structure des variables globales*/

  int monRang;           /* rang du processus */

  char host[20];         /*le nom de la machine sur lequel un processus tourne*/

 /*allocation mémoire*/
  c = (struct shareddata *)malloc(sizeof(struct shareddata));

  // INITIALISATION DE L'ENVIRONNEMENT MPI
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &monRang);
  MPI_Comm_size(MPI_COMM_WORLD, &c->nbrProc);

  // VERIFICATION DES PARAMETRES
  if (!monRang){
    if (argc != 4){
      fprintf(stderr,"Utilisation testSumV<x> <Nombre d'opérations addition par
              itération  par esclave> <Nombre d'itération><Nombre itération entre
             deux tests de migration> \n");
      MPI_Abort(MPI_COMM_WORLD, 1);
      return 0;
    }
  }

  // INITIALISATION DES PARAMETRES
  c->m = atof(argv[1]);
  c->n = atof(argv[2]);
  c->periodLB = atoi(argv[3]);

  // DEBUT DU PROGRAMM
  if (!monRang)
    debut(c);

  // PARTIE MAITRE
  if (!monRang){
    maitre(c);
  }

  // PARTIE ESCLAVE
  else{
    esclave(c,monRang);
```

```
  }

  // FIN DU PROGRAMM
  gethostname(host, 20);
  printf("fin processus %d sur %s\n",monRang,host);

  if (!monRang){
    fin();
  }

  // FERMETURE DE L'ENVIRONNEMENT MPI
  MPI_Finalize();
  //return 0;

} /*main*/
```

Annexe E

Contenue du CD

Le CD accompagnateur[1] contient les répertoires suivants :

- rédaction
  - mémoire en format pdf, post-script et dvi
  - source latex
- programme *sum*
  - code en version C séquentiel, MPI et AMPI
  - résultats des tests
- programme *press*
  - code en version C séquentiel d'origine, C séquentiel structuré, C séquentiel parallélisé, MPI et AMPI
  - résultats des tests

[1] Voire aussi `http://users.skynet.be/teller/memoire/`.

Glossaire

AMPI : AMPI (Adaptive MPI) est une implémentation du standard MPI qui se base sur CHARM++ et CONVERSE pour fournir des optimisations adaptatives.

Application dynamique : Une application dynamique est une application qui évolue pendant l'exécution. A l'inverse l'exécution d'une application statique est déterminée pendant la compilation.

Application irrégulière : Une application irrégulière est une application qui ne peut pas être structurée en parties semblables et/ou qui évolue dynamiquement pendant l'exécution (voir *application dynamique*).

Architecture parallèle : Une architecture parallèle est une architecture informatique (niveau hardware) qui permet de calculer plus qu'une tâche à un moment donné. Cela ne nécessite pas nécessairement plusieurs processus, les processus actuels sont capables d'effectuer plusieurs opérations dans un même cycle d'horloge.

Athapascan : Athapascan est un bibliothèque parallèle des langages C et C++ qui est basée sur le *multi-threading* et qui fournit des optimisations pour le partage des données.

Calcul parallèle : Le calcul parallèle caractérise les calculs sur ordinateur effectué dans le domaine de l'informatique parallèle.

Calcul à haute performances : Le calcul à haute performance est souvent un calcul parallèle qui a pour objectif de faire un très grand nombre d'opérations de calcul dans un minimum de temps.

Charge : Sous le terme charge nous classifions tous les effets en terme de ressource de calcul (CPU), utilisation de mémoire ou encore des entrées et sorties, que le traitement des tâches peuvent produire.

CHARM : CHARM est une librairie pour le langage C qui se base sur le *message-driven*.

CHARM++ : CHARM++ est une librairie orienté-objet de programmation parallèle pour C++ portable sur la plus part des machines parallèles. CHARM++ qui se base sur le *message-driven*, fourni avec l'aide de CONVERSE un ensemble d'optimisation adaptative.

Cilk : Cilk est une extension parallèle du langage C basé sur le *multi-threading*.

Cluster : Un cluster ou grappe est une architecture parallèle d'un ensemble de noeuds (ordinateur ou machine SMP) d'un caractère homogène qui sont interconnectés par un réseau local et rapide.

Communication collective : Une communication collective dans le domaine parallèle concerne une communication entre tous ou un sous-ensemble de tâches d'un programme parallèle. Il existe des communications collectives avec plusieurs émetteurs et un destinataire ou avec un émetteur et plusieurs destinataires.

Communication point à point : Une communication point à point dans le domaine parallèle concerne uniquement une communication entre deux tâches bien identifiées (un émetteur et un destinataire) d'un programme parallèle

Condition au bord périodique : La condition au bord périodique est un mécanisme de la dynamique moléculaire qui permet de dupliquer un espace de simulation afin de former un espace infini pour exclure des effets de bord.

CONVERSE : Converse est un support d'exécution parallèle qui fournit des optimisations adaptatives comme l'équilibrage dynamique de charge.

Dynamique moléculaire : La dynamique moléculaire est une technique de simulation numérique (sur ordinateur) qui permet d'étudier les propriétés d'une matière au niveau moléculaire ou atomique pour en déduire des propriétés macroscopiques.

Environnement : Un environnement est un ensemble d'outils principalement software qui fournissent des services élevés du genre système d'exploitation , comme par exemple l'équilibrage de charge dynamique.

Equilibrage : L'équilibrage est l'essai de transformer une distribution non-uniforme de quelque chose dans une distribution uniforme. Dans l'informatique, on essaye souvent d'équilibrer l'utilisation des ressources.

Equilibrage de charge : L'équilibrage de charge (*Load Balancing*) est un mécanisme qui permet de répartir le plus équitablement possible la charge entre différents sous-systèmes. Les objectifs principaux de l'équilibrage des charges consistent à minimiser le temps de réponse moyen du système et à maximiser le degré d'utilisation des ressources.

Grid : Un grid ou grille est l'interconnexion "ouverte" via un réseau (par exemple internet) d'un ensemble important de ressources (puissance de calcul, mémoire, ...) souvent hétérogènes pour mettre ces ressources à disposition des acteurs de la grille. Ainsi ces acteurs puissent effectuer des tâches qui étaient impossibles ou difficiles avec les ressources d'un seul acteur.

Heap : Le heap ou tas reprend les variables allouées dynamiquement (par exemple les pointeurs) par un programme ou une sous-routine.

HPF : HPF (High Performance Fortran) est une extension parallèle du langage Fortran via des directives compilateurs pour le parallélisme des données.

Informatique parallèle : Le parallélisme ou informatique parallèle dénote la possibilité d'exécution plus d'une tâche à un moment donné sur une infrastructure informatique.

Infrastructure parallèle : L'infrastructure parallèle comprend l'architecture parallèle ainsi que ses outils software pour accomplir les objectifs de l'informatique parallèle.

Langage parallèle : Le langage parallèle est un langage de programmation qui permet d'écrire des programmes en utilisant des techniques de programmation parallèle.

Lennard-Jones : La force de Lennard-Jones exprime l'accélération qu'un atome subit à cause des interactions avec les atomes qui l'entourent.

Message-Driven : Le message-driven est un modèle de programmation parallèle qui se base sur l'invocation de calcul à distance via l'envoi d'un message.

Message-Passing : Le message-passing est un modèle de programmation parallèle qui se base sur la coordination par échange de message.

Migration : La migration d'une tâche est définie comme un changement de l'endroit de l'exécution de cette tâche pendant son exécution. La migration fait partie d'une stratégie d'équilibrage dynamique des charges.

MIMD : (Multiple Instruction Multiple Data) Une architecture de la classification de Flynn qui traite des flots d'instructions avec leurs flots de données réciproques.

MMP : (Massivly Parallel Porcessor) Une architecture parallèle réelle qui est un multi-processeurs où tous les processeurs disposent une zone de mémoire locale (MIMD-DM (*Distributed Memory*)).

MPI : MPI (Message Passing Interface) est un standard qui spécifie le paradigme *message-passing*.

Mpich : Mpich est une implémentation *freeware* du standard MPI.

Multi-Threading : Le mutli-threading est une technique qui permet d'exécuter simultanément plusieurs flux de contrôle sur une architecture séquentielle ou parallèle.

OpenMP : OpenMP est une extension des langages C et Fortran qui se base sur le partage des données et le *multi-threading* sur une architecture à mémoire partagée.

Ordonnancement : L'ordonnancement des tâches consiste à ordonner en temps et lieu les différentes tâches avant le début de leurs traitements de manière à satisfaire les objectifs d'équilibrage de charge.

Parallélisme : voir *Informatique parallèle*.

Persistance : Le principe de persistance indique que les mesures de l'exécution passée d'une tâche sont suffisantes pour prédire le comportement future de la tâche.

PM2 : PM2 est un support d'exécution parallèle qui fournit des optimisations pour la communication et pour l'équilibrage de charge pour des applications *multi-thread*.

Police : Une police d'équilibrage de charge décide selon quelle politique prioritaire et sur la base de quelles informations pertinentes des charges sont à distribuer.

Potentiel : Le potentiel est une énergie qu'un corps dispose uniquement à cause de sa position.

Processus : Un processus correspond à l'exécution d'une tâche sur un processeur. Souvent il dispose de sa propre mémoire locale de travail.

Programmation parallèle : La programmation parallèle comprend les techniques et méthodes de programmation dans l'informatique parallèle. Elle vise à diminuer le temps d'exécution pour résoudre un problème ou à pouvoir augmenter la taille d'un problème.

Prédiction-Correction : La prédiction-correction est un algorithme utilisé pour simuler la dynamique d'un système dans une simulation de la dynamique moléculaire.

PVM : PVM (Parallel Virtual Machine) est une bibliothèque de *message-passing* qui se base sur la construction d'une machine virtuelle qui cache l'architecture paralléle réelle.

Rayon de coupure : Le rayon de coupure est la distance minimale de deux atomes pour lequel leur force interaction est négligeable au niveau d'un calcul physique.

Ressource : Sous le terme de ressource on classifie dans ce travail la puissance de calcul disponible, la taille de la mémoire, le nombre et le délai des entrées et sorties ou le débit d'un réseau.

SMP : (Symetric MultiProcessor) Une architecture parallèle réelle qui est un multi-processeurs où tous les processeurs se partagent la même zone de mémoire (MIMD-SM (*Shared Memory*)).

SPMD : (Singel Programm Multiple Data) C'est le modèle d'une architecture parallèle qui correspond à l'exécution des copies d'un même programme sur plusieurs flots de données.

Stack : Le stack ou pile reprend les variables locales d'un programme ou d'une sous-routine.

Support d'exécution : Voir *environnement*.

Système d'équilibrage de charge : Un tel système est un gestionnaire qui dirige, par exemple un architecture parallèle, selon les principes d'équilibrage de charge (voir équilibrage de charge)

Système parallèle : Un système parallèle est un environnement qui fournit des services utiles en calcul parallèle.

Sérialisation : La sérialisation de données correspond à la transformation d'une structure de données dans un format qui peut être transmis par un réseau.

Temps de réponse : Le temps de réponse d'un ordinateur ou tout autre système d'information est le temps entre le lancement d'une tâche sur le système et la fin de la dernière avec une production éventuelle d'un ou plusieurs résultats.

Thread : Un *thread* est un flot d'instructions qui s'exécutent en manipulant des données stockées dans une mémoire commune. Un thread peut être vu comme un processus simple sans mémoire locale.

Tâche : Une tâche pour un système ou infrastructure informatique est tout un ensemble d'opérations que le système peut accomplir pour satisfaire un besoin de l'utilisateur ou du système lui-même (par exemple une autre tâche). Une tâche peut par exemple être une simple transaction de quelques microsecondes ou encore un calcul mathématique qui peut prendre plusieurs semaines.

User-level thread : Un *user-level thread* est un *thread* qui est traité au niveau d'une application au contraire d'un *kernel thread* qui dépend de la gestion du système exploitation.

Variable globale : Une variable globale est une variable dont la portée est tout le programme.

Variable locale : Une variable locale est une variable dont la portée est limitée à une sous-routine.

Virtualisation : La virtualisation des processeurs est le fait d'affecter un processus qui demande en exclusivité un processeurs réel à un processeur virtuel et de placer alors plusieurs de ces processeurs virtuels sur un processeur réel afin de profiter du *multiprocessing*.

Workstealing : La méthode de *Workstealing* (vol de travail) incite un processeur qui est inoccupé à demander du travail chez d'autres processeurs.

Index