Parallèlisme massif et langage à objets : une approche SPMD

Jean-Marc Jézéquel

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1 9 9 2

25ème anniversaire

N° 1607

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

PARALLELISME MASSIF ET LANGAGE A OBJETS : UNE APPROCHE SPMD

Jean-Marc JÉZÉQUEL

Février 1992

RR-1607

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Parallélisme massif et langage à objets : Une approche SPMD

Jean-Marc JÉZÉQUEL
E-mail: jezequel@irisa.fr

Publication Interne n° 623 - Décembre 1991
26 pages - Programme 1

Résumé

La plupart des langages à objets (LAO) parallèles utilise un modèle général de parallélisme fondé sur la notion de processus séquentiels communiquant, à la CSP. Dans le contexte du calcul scientifique intensif, cette approche rend difficile la programmation efficace de machines massivement parallèles. C'est pourquoi nous proposons d'adopter pour ce type de langage une autre forme de parallélisme, connue sous le nom de parallélisme de données. Nous décrivons comment ce parallélisme de données peut être intégré dans un LAO séquentiel de manière simple et élégante —en utilisant uniquement les constructions du langage— afin d'exploiter la puissance potentielle des machines massivement parallèles à mémoires distribuées. Nous présentons ensuite EPEE, notre Environnement Parallèle d'Exécution de Eiffel, avec un exemple de son utilisation sur un paradigme bien connu dans le domaine du parallélisme (calculs sur des matrices). Après quelques remarques sur notre implantation, nous donnons quelques résultats expérimentaux de performances, et nous concluons sur la généralité de notre approche.

Massive Parallelism and Object Oriented Language: an SPMD approach

Abstract

Most parallel object oriented languages (OOL) are currently using a general parallelism model based on communicating sequential processes. This approach makes it difficult to program massively parallel systems in an easy and efficient way. So we propose to use another form of parallelism, known as data parallelism. We describe how a sequential OOL can embed data parallelism in a clean and elegant fashion —without language extensions— in order to exploit the potential power of massively parallel systems. Then we present EPEE (an Eiffel Parallel Execution Environment) at work with a well known parallel paradigm (matrix computations), along with experimental performance results. We draw some conclusions on the generality of this approach.

1 Introduction

Le calcul scientifique intensif va de plus en plus s'appuyer sur des architectures de machines massivement parallèles à mémoires distribuées (APMD dans la suite), seules issues possibles dans la course vers les Teraflops (cf. les projets américains et européens Touchstone et GP-MIMD). En effet, seules des architectures multiprocesseurs modulaires peuvent être étendues afin de fournir une puissance de calcul —théoriquement— illimitée.

Il est maintenant devenu un lieu commun de dire que le principal facteur qui limite encore la diffusion des APMD concerne leur programmation. En effet, écrire (ou porter) des programmes d'application sur ces architectures reste une tache ardue et coûteuse. L'environnement logiciel des APMD aujourd'hui disponibles est en général assez pauvre : il consiste principalement en un ensemble de librairies de routines pour gérer la communication entre processus décrits dans des langages séquentiels comme FORTRAN, C ou Lisp. Pour utiliser une APMD, un programmeur doit alors posséder une bonne maîtrise à la fois de son domaine d'application, des méthodes de parallélisations de son architecture et de son système d'exploitation. Le code finalement obtenu est en outre difficile à développer, comprendre, mettre au point et maintenir.

Aussi de nombreux auteurs se sont-ils attachés à développer de nouveaux langages en cherchant à y intégrer proprement les aspects liés au parallélisme. Parmi les différentes approches possibles, celles fondées sur la programmation par objets paraissent particulièrement intéressantes, peut-être à cause des analogies apparentes qui existent entre des processus et des objets communiquant par messages.

Dans les langages à objets, un objet est une unité de structuration de programme encapsulant données et méthodes (procédures et fonctions) travaillant sur ces données. En utilisant la terminologie Smalltalk, on dit que des objets communiquent par envoi de messages.

On voit donc apparaître un certain nombre de points communs avec la notion de processus. La tentation est donc forte d'intégrer ces deux notions dans celle d'objet actif. POOL-T [1] est certainement un des exemples les plus significatifs illustrant cette approche : une fois créé, un objet peut continuer à être actif après avoir rendu la main à son créateur. La communication entre objets est ici de type Remote Procedure Call (appel de procédure à distance), et le programmeur contrôle alors explicitement le parallélisme et l'accès aux données.

Un autre moyen pour introduire du parallélisme dans les langages à objets consiste à rendre asynchrones les appels aux méthodes des objets : un objet appelant la méthode d'un autre objet peut continuer son activité en parallèle avec l'objet exécutant la méthode appelée. Ceci est implanté par exemple dans ABCL/1 [15], ELLIE [2], ou encore ConcurrentSmalltalk [14].

Ces deux idées d'objet actif et d'appels de procédure asynchrones sont utili-

sées pour créer des LAO parallèles soit en intégrant ce parallélisme dès la phase de conception d'un nouveau langage de programmation concurrent (ELLIE, POOL-T, ABCL/1, etc.), ou plus simplement en étendant des langages séquentiels déjà existants pour leur permettre de gérer le parallélisme, comme dans Distributed-Smalltalk [4] qui étend Smalltalk ou COOL [6] qui étend C++.

Ainsi, presque tous les LAO parallèles sont fondés sur un modèle de programmation MIMD (Multiple Instructions, Multiple Data) très dynamique, qui semble bien convenir à certains types de problèmes où le parallélisme pré-existe (systèmes d'exploitation distribués, contrôle de processus industriels, etc.) et où les problèmes se posent en termes de coopération. Ce modèle de programmation MIMD est en effet particulièrement bien adapté à la gestion d'un parallélisme de type fonctionnel : une fonction est décomposée en sous-fonctions pouvant être exécutées par de nouveaux flots de calculs. Une application est alors découpée en un ensemble de processus communicants dont la structure dépend de l'application et de la façon dont l'utilisateur la décompose. Comme ces processus sont par nature hétérogènes, des problèmes d'équilibrage de charge doivent être réglés soit par le système d'exploitation (ce qui est coûteux) soit par l'utilisateur lui-même.

D'autre part, la structure des communications entre ces processus est entièrement gérée par le programmeur, qui est donc confronté de plein fouet à tous les problèmes bien connus de la communauté des concepteurs de protocoles de communications : interblocages (dead locks), absence d'état global rendant difficile la détection de la terminaison par exemple... Le débogage de ce type de programme est particulièrement malaisé, car si les parties séquentielles peuvent être déboguées à l'aide de débogueurs séquentiels spécialement modifiés pour pouvoir s'exécuter sur APMD, le débogage des aspects coopération entre processus reste en revanche un problème ouvert. En effet, par exemple l'ajout d'un simple ordre de trace peut modifier radicalement le comportement d'un tel programme.

Mais un des aspects sans doute les plus importants de l'utilisation d'APMD concerne les problèmes d'efficacité. En effet il ne faut pas perdre de vue que l'obtention de performances extensibles linéairement en fonction du nombre de processeurs d'une APMD est la principale motivation de l'utilisation de machines parallèles comportant un grand nombre de processeurs. Or un parallélisme de type fonctionnel n'est pas extensible à volonté: étant lié à la structure dynamique d'une application, peu de structures régulières peuvent être dérivées du programme original lors de la compilation, aussi la plus grande partie du travail doit être menée à l'exécution. Cette nature dynamique implique l'utilisation de mécanismes généraux et coûteux pour permettre par exemple la création dynamique de processus, le nommage uniforme des objets, la migration de données ou l'appel de méthodes à distance, l'équilibrage de charge, le scheduling, etc.

Pour surmonter ces problèmes, nous proposons d'utiliser un modèle de parallélisme différent, connu sous le nom de *parallélisme de données* (Data Parallelism).

Cette approche semble en effet assez naturelle dans le cadre de la programmation orientée objet, qui se focalise traditionellement plutôt sur les données que sur les fonctions. Quelques principes pour construire un LAO gérant le parallélisme de données ont été introduites dans [10], et les idées de bases pour étendre C++ dans le même sens ont été présentées dans [7]. Avec EPEE, notre Environnement Parallèle d'Exécution de Eiffel, nous voulons aller encore plus loin en intégrant totalement la distribution de données et le parallélisme afférant dans un LAO séquentiel, tout en n'utilisant que les constructions disponibles dans le langage. EPEE est fondé sur Eiffel [11], car c'est un LAO qui offre tous les concepts dont nous avons besoin avec une syntaxe et une sémantique claires et précises. Mais notre approche ne dépend pas particulièrement de ce langage ; elle pourrait être utilisée avec n'importe quel LAO disposant d'une encapsulation stricte, de l'héritage multiple, de la liaison dynamique et d'une certaine forme de généricité.

Nous montrons dans le paragraphe suivant que ces concepts sont suffisament puissants pour permettre l'intégration d'un parallélisme de données dans un LAO séquentiel de manière simple et élégante, sans en altérer ni la syntaxe ni la sémantique. A l'aide d'un exemple bien connu dans le domaine du parallélisme, nous présentons au paragraphe 3 notre approche depuis la spécification et l'analyse du problème jusqu'à son implantation, ainsi que quelques résultats expérimentaux de mesures de performances. Dans la dernière partie, nous tirerons quelques conclusions sur notre approche et indiquerons quelques perspectives.

2 Intégration d'un parallélisme de données dans un LAO

2.1 Un modèle de programmation pour exploiter le parallélisme de données

La quasi-totalité des langages à objets parallèles est fondée sur un modèle de programmation MIMD pour les APMD, difficile à manipuler dans un cadre de parallélisme massif. Mais d'autres modèles de programmation s'éloignant moins des habitudes des programmeurs ont été proposés par ailleurs pour les APMD : par exemple le modèle SIMD (Single Instruction, Multiple Data) permet parfois de décrire certains algorithmes de manière naturelle ; et des machines SIMD ont d'ailleurs été construites pour l'implanter efficacement (par exemple la Connection Machine). D'importants travaux de recherche sont aussi menés sur le thème de la parallélisation totalement automatique de codes SISD (Single Instruction Single Data) de type FORTRAN. Mais cette approche ne semble pas très bien adaptée aux APMD : quelque soit le nombre de processeurs d'une APMD, il est rare de voir des accélérations (speed-up) supérieures à 10 pour la parallélisation de programmes FORTRAN déjà existants (dusty-deck parallelization).

Le modèle SPMD (Single Program, Multiple Data) est une sorte de compromis entre ces deux approches, qui prend du modèle SISD la simplicité conceptuelle et du modèle SIMD le parallélisme. Ce modèle tente de prendre en compte le fait que la plupart des problèmes qu'on souhaite résoudre sur des APMD est caractérisée par la grande quantité de données à traiter. L'idée de base (décrite dans [5]) est alors de partitionner cette masse de données, et d'associer statiquement un processus à chacune de ces partitions. Chaque processus exécute le même programme (correspondant au programme utilisateur initial), mais seulement sur sa partition. Ainsi la vue qu'a cet utilisateur de son programme reste séquentielle : une séquence d'actions est appliquée à un ensemble de données ; et le parallélisme découle de la décomposition des données, ce qui conduit à une forme de parallélisme régulière et extensible.

L'intérêt de ces modèles pour la programmation d'APMD est de masquer à l'utilisateur du modèle la structure réelle de l'APMD et notamment les mécanismes d'échanges de messages. Dans Pandore [3] par exemple, l'utilisateur décrit la répartition de ses données sur des processeurs, puis programme son application dans un modèle purement séquentiel. Le compilateur transforme ensuite ce code en un programme SPMD : chaque processeur exécute le même programme (mais pas forcément la même instruction) sur des parties différentes des données. De manière similaire, ce processus de parallélisation est mené dans SUPERB [16] grâce à un dialogue interactif avec le compilateur.

Mais de toutes façons, ce type de compilateur doit contenir une grande expertise des problèmes et algorithmes parallèles sur les données qu'il manipule (qui sont pour l'instant limitées à des tableaux) afin de produire du code efficace.

C'est pourquoi nous proposons de sortir (conceptuellement) cette expertise du cœur du compilateur pour l'encapsuler avec les données qu'elle manipule. Nous nous proposons de montrer que l'utilisation d'un langage à objets permet de réaliser proprement cette encapsulation afin de laisser à l'utilisateur l'illusion d'une programmation séquentielle.

2.2 Encapsulation du parallélisme

Différents projets de recherche ont eu l'idée d'utiliser les possibilités d'encapsulation et d'abstraction de données existant dans les LAO pour encapsuler une certaine forme de parallélisme. Par exemple dans [13] il est proposé d'étendre C++ à l'aide de *path expressions*, sortes de spécifications d'ordonnancement d'accès concurrents aux méthodes d'un objet. Nous voulons ici aller plus loin, en proposant de complètement cacher tout parallélisme à l'utilisateur : nous rejettons l'idée que la structure des communications entre processus soit claquée sur le paradigme de la communication par messages entre objets. Ainsi, de même que lorsqu'on encapsule dans une classe *Liste_Chainée* (voir figure 1) les opérations non triviales de manipulations de pointeurs afin de produire un module réutilisable (sans avoir justement à se soucier

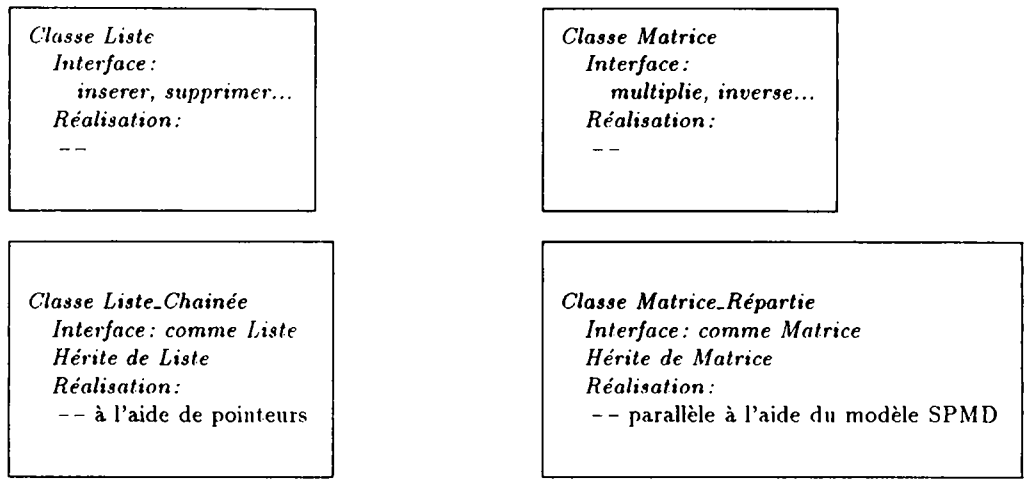

Figure 1 : Analogie entre encapsulation du parallélisme et des opérations sur les pointeurs

de pointeurs) nous pensons qu'il faut encapsuler dans des classes *had hoc* les opérations de communication du modèle SPMD afin de fournir des classes réutilisables ayant des interfaces purement séquentielles. Nous montrons ici que notre approche peut être totalement exprimée en utilisant les constructions d'un langage à objets possédant l'héritage multiple, et en s'appuyant sur ses mécanismes de base comme la modularité, l'abstraction et l'encapsulation de données, et la liaison dynamique. Eiffel et C++ sont deux exemples de tels langages.

Comme nous l'avons expliqué ci-dessus, nous nous intéressons principalement à des classes agrégeant de grandes quantités de données. Pour une telle classe C, nous voulons fournir une implantation répartie (et donc parallèle) que nous appelerons *Distributed_C*, ayant exactement la même interface (séquentielle) que C. Dans un cadre de LAO, nous pouvons réaliser cela en utilisant le concept d'héritage: on dira que *Distributed_C* hérite de C, c'est à dire que c'est une spécialisation de C.

Nos classes réparties ayant un certain nombre de points communs (leurs instances sont des agrégats répartis), nous factorisons et matérialisons ces points communs grâce à une classe *had hoc* que nous appelons *Distributed_Aggregate*. Utilisant le concept d'héritage multiple, nous faisons alors en sorte que la classe *Distributed_C* hérite à la fois de C et de *Distributed_Aggregate* (cf. figure 2).

Au cœur de l'EPEE, cette classe *Distributed_Aggregate* n'est pas sans corrélation avec la notion proposée dans [8] : c'est un agrégat de données génériques réparties sur une APMD, encapsulé avec un ensemble de méthodes pour accéder aux données de manière transparente, pour les redistribuer, pour exécuter une méthode sur

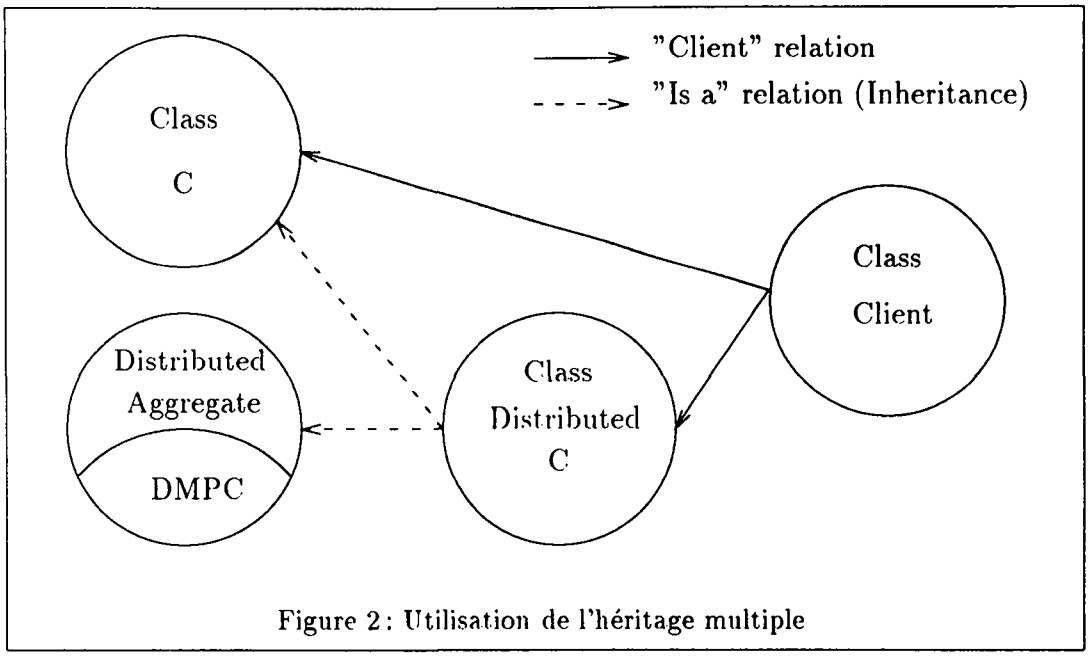

Figure 2 : Utilisation de l'héritage multiple

chacun de ses éléments, et pour calculer toute fonction associative sur l'agrégat. La façon dont ces méthodes sont implantées dépend en fait de l'architecture sous-jacente ; aussi pour préserver une portabilité totale au niveau du LAO considéré, nous introduisons un module d'interface dépendant du système entre l'APMD et la classe *Distributed_Aggregate*, comme illustré en figure 3.

Voici schématiquement comment on peut implanter ces concepts sur une APMD constituée d'un ensemble de processeurs ne communiquant que par échanges de messages. Comme un site donné ne possède qu'un sous-ensemble des données de l'agrégat réparti, nous devons tout d'abord mettre en œuvre un mécanisme permettant l'accès transparent aux données distantes, tout en préservant la sémantique d'un accès local. Ce mécanisme est fondé sur la notion de Refresh/Exec (décrite dans [3]) : une donnée est rafraichie *(Refresh)* avant toute opération de lecture, et l'écriture n'est prise en compte que pour les données locales (principe de l'écriture locale : *Exec*). Sur une telle APMD, une implantation possible du *Refresh* est la suivante :

```
If Owner(V) = this_node
  then broadcast (V) and return (Value of V)
  else receive_from (Owner(V),V) and return (Value of V)
```

Pour implanter la redistribution et la circulation des données, on peut utiliser le mécanisme de communication FIFO par messages de l'APMD, et pour les méthodes

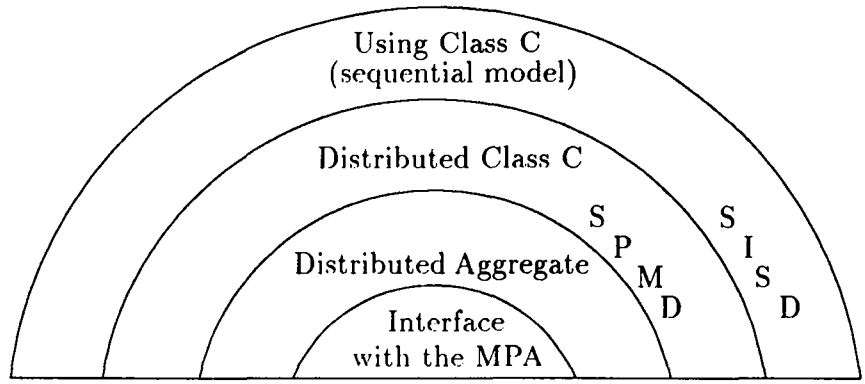

Figure 3 : EPEE: les différents niveaux de programmation d'une APMD

globales, on peut soit les implanter en utilisant des algorithmes distribués de la littérature, soit utiliser les opérations globales déjà disponibles sur certaines machines comme les hypercubes Intel iPSC.

2.3 Utiliser EPEE pour la parallélisation d'applications orientées objets

Nous distinguons deux niveaux de programmation dans notre modèle : le niveau utilisateur (ou client) des classes parallélisées et le niveau concepteur (fournisseur) de ces classes. Notre objectif est qu'au niveau utilisateur ne transparaisse de l'exécution sur APMD qu'un gain de performance, si possible proportionnel au nombre de processeurs employés. Le concepteur d'une classe parallélisée aura quant à lui un travail beaucoup plus important à effectuer : à partir de la spécification abstraite (ou d'une implantation séquentielle) d'un objet, concevoir des schémas de distribution de ses données et de ses algorithmes en assurant à la fois leur portabilité, leur efficacité, et l'extensibilité de leurs performances[1].

Cette approche autorise une migration incrémentale et modulaire de logiciel séquentiel vers un environnement massivement parallèle à mémoire distribuée : en résumé, pour chaque classe qu'on pense pouvoir paralléliser avantageusement (qui agrège typiquement une grande quantité de données), il suffit dans un premier temps de créer un descendant héritant de la classe agrégat réparti, puis de définir les politiques de répartition de l'agrégat. En utilisant les abstractions fournies par la classe agré-

[1] Notons que dans les approches par compilation, ce travail est à la charge du concepteur du compilateur, qui doit en plus être capable de "reconnaître" les structures à paralléliser.

gat réparti de EPEE, on assure la correction des accès aux données en redéfinissant les méthodes d'accès. Finalement, chaque méthode critique (en terme d'efficacité) peut être redéfinie pour tirer parti du modèle de programmation SPMD fourni dans EPEE.

Ainsi un objet O pourrait être déclaré de type statique C, et instancié avec le type dynamique C_réparti. O continue alors a être utilisé comme une instance de C bien que certaines de ses méthodes soient redéfinies en tenant compte de son caractère réparti, soit pour préserver sa sémantique, soit pour en tirer parti afin de gagner des performances. Donc grâce au mécanisme d'encapsulation, un client de la classe C_répartie ne verra comme seul changement qu'un gain de performance pour certaines méthodes.

Les résultats attendus de cette manière d'introduire le parallélisme sont concomittants de l'approche orientée objet : qualité du logiciel, réutilisabilité, réduction des temps et coûts de développements, de mise au point et de maintenance.

Dans le paragraphe suivant, nous appliquons cette méthode à un problème bien connu dans la communauté du parallélisme : des calculs sur des matrices.

3 Application aux calculs sur des matrices

3.1 Analyse d'un problème

La parallélisation de calculs d'algèbre linéaire est un paradigme de programmation parallèle. C'est pourquoi nous présentons notre approche au travers de cet exemple, afin de permettre au lecteur de se concentrer plutôt sur l'approche que sur les particularités de l'exemple. Mais l'intérêt de cette approche réside bien sûr dans le fait qu'elle n'est pas limitée à ce type d'application.

Soit la classe *Matrix* une classe encapsulant le type abstrait matrice de réel et présentant l'interface représentée en figure 4.

Cette classe peut être utilisée par le programme suivant :

```
Class client
feature Create is
    local M, A, B, C: Matrix
    do
        A.Create (512, 512); B.Create (512, 320); C.Create (320, 512);
        A.read; B.read; C.read;
        M:= A + B * C;
        M.inversion;
        M.print;
    end;
end;
```

Selon la méthode définie au paragraphe précédent, commençons par analyser ce programme en détail, en étudiant les moyens de gagner des performances.

```
class interface MATRIX exported features
        read, infix "+", infix "*", inversion, nb_rows, nb_columns, item, put
feature specification
        Create (nb_rows, nb_columns: INTEGER)
                require
                        not_flat: nb_rows > 0; not_thin: nb_columns > 0
        Read            -- Read Matrix elements
        infix "+" (m: like Current): like Current
                require
                        m_must_exist: not m.Void;
                        compatible: nb_rows=m.nb_rows
                                    and nb_columns=m.nb_columns
        infix "*" (m: like Current): like Current
                require
                        m_must_exist: not m.Void;
                        compatible: nb_columns=m.nb_rows
        Inversion       -- compute Result such that Result*Current=Identity
                require
                        square: nb_rows=nb_columns
end interface -- class MATRIX
```

Figure 4 : La classe *Matrix* en Eiffel

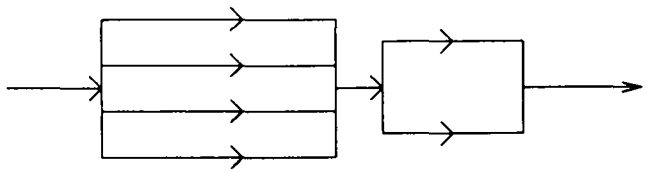

Figure 5 : Représentation graphique du modèle BSP

La première ligne contient trois instructions de création, chacune de complexité $0(1)$ en fonction de la taille n des matrices. Ces trois instructions sont totalement indépendantes et pourraient donc être exécutées en parallèle. Cependant ce type de parallélisme ne nous intéresse pas : en effet, d'une part rien ne prouve que l'exécution en parallèle de ces trois instructions soit plus rapide que leur exécution séquentielle (à cause de l'overhead dû aux communications et au démarrage des tâches chargées de les exécuter) ; et d'autre part ce parallélisme n'est pas extensible (scalable) ni en fonction de la taille des matrices, ni en fonction du nombre de processeurs de l'APMD.

Ce même raisonnement est valable pour la seconde ligne (la saisie des matrices). Cependant, ici chaque instruction est en $0(n^2)$, et il pourrait être intéressant de la paralléliser en faisant en sorte que chaque processeur ne lise qu'une partie des données. Si p est le nombre de processeurs de l'APMD, un speed-up de p est alors théoriquement possible, à condition toutefois que le matériel permette effectivement l'accès concurrent à des périphériques d'entrées-sorties (cas du module CFS de Intel pour l'iPSC/2). Comme on le voit, ce parallélisme est extensible.

La troisième ligne contient deux opérations (addition et multiplication de matrices) pour lesquelles on sait qu'il existe des algorithmes parallèles ayant des speed-up théoriques de p, donc extensibles. L'inversion de matrice de la ligne suivante peut elle aussi être parallélisée de manière extensible. Enfin l'écriture du résultat sur un support séquentiel (imprimante, écran, fichier) est une opération séquentielle sauf si on dispose de matériel spécial (par exemple disque à têtes multiples indépendantes, ou structure de fichier répartie).

3.2 Objet fragmenté et modèle SPMD

N'importe quel programme n'est pas "massivement" parallélisable (ou parallélisable de manière extensible), *i.e.* sa parallélisation n'apportera pas (ou peu) de gain de performance sur une APMD comportant un grand nombre de processeurs. Valiant a proposé dans [12] une théorie pour rendre compte de cela : le modèle BSP (Block Synchronous Parallel). Un programme entre dans le cadre BSP s'il peut être vu comme une succession de phases parallèles séparées par des barrières de synchro-

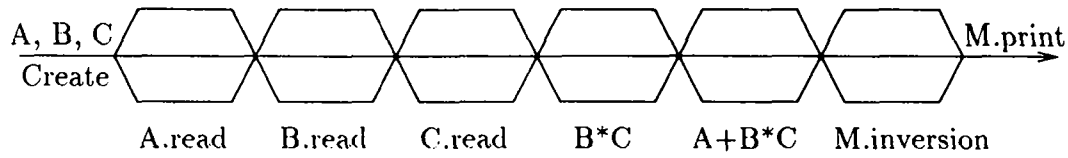

Figure 6 : Exécution SPMD de l'application *Class Client* sur trois processeurs

nisation (voir figure 5) et des phases séquentielles. Si l'on considère l'APMD comme le co-processeur de son frontal, seules les phases parallèles s'y exécutent tandis que les phases séquentielles s'exécutent sur le frontal, qui est aussi chargé de la synchronisation. Si l'APMD ne possède pas de frontal (donc est vue comme une machine autonome, ce qui est la tendance pour les futures architectures) il suffit de répliquer l'exécution du programme frontal sur chacun des processeurs de l'APMD : c'est la solution que nous avons choisie.

Un calcul n'est alors efficacement parallélisable que si le coût des synchronisations et des communications et autres traitements dûs au parallélisme est compensé par le gain de performances apporté par la parallélisation de l'algorithme.

Comme l'illustre la figure 6, notre exemple de classe client entre donc parfaitement dans le modèle BSP de parallélisme extensible. On n'obtiendra cependant un speed-up supérieur à 1 que si la taille des matrices est suffisamment grande, et ce speed-up sera d'autant plus important que le rapport temps de calcul sur temps de communication sera grand.

Ayant maintenant mis en évidence le type de parallélisme qui nous intéresse, i.e. parallélisme intra opération sur l'agrégat plutôt que parallélisme inter-instructions du programme client, étudions en détail la façon de le réaliser au sein de la classe matrice répartie.

3.3 Distribution de données dans la classe Matrice

Notre point de départ est la classe matrice et la classe agrégat réparti (*Distributed_Aggregate*). A partir de cela, nous créons une classe matrice répartie (*Distributed_Matrix*) ayant la même interface que *Matrix*, et héritant des deux :

```
Class Distributed_Matrix
    export repeat Matrix -- Same interface than Matrix
    inherit Distributed_Aggregate; Matrix -- inherit from both ancestors
```

Le constructeur de la classe Matrice Répartie (méthode *Create* en Eiffel) est défini de manière à découper la matrice en parties de tailles égales allouées à chacun

des sites de l'APMD, en utilisant les fonctionnalités de la classe agrégat réparti. Pour créer effectivement la matrice, chaque site de l'APMD exécute l'instruction de création (c'est pourquoi on appelle ce modèle de programmation SPMD), et réserve un espace mémoire pour stocker son morceau de matrice.

Dans le cas d'une matrice, de nombreuses politiques de répartition ont été proposées : par lignes ou colonnes entrelacées, par blocs de lignes, par blocs de lignes entrelacées, par blocs... Nous devons donc envisager la possibilité de re-distribuer notre agrégat en fonction de la méthode invoquée, ce qui implique que l'agrégat possède des attributs décrivant sa répartition, et des méthodes permettant de passer d'une répartition à une autre. Pour notre exemple, nous sommes intéressés par des répartitions en lignes ou en colonnes entrelacées et en blocs de lignes ou colonnes, éventuellement entrelacés. Nous avons donc besoin de deux attributs: un booléen indiquant si on est en lignes ou en colonnes, et un entier r donnant le nombre de lignes (ou de colonnes) par bloc. Autrement dit, pour une matrice à n lignes et une APMD à p processeurs, l'interprétation des valeurs de r est la suivante:

- si r=1, on a une répartition par lignes (ou colonnes) entrelacées
- si r=E$\left(\frac{n+p-1}{p}\right)$ on a une répartition par bloc
- $1 <$ r $<$E$\left(\frac{n+p-1}{p}\right)$ on a une répartition par blocs entrelacés.

Par convention, ajoutons que r=0 signifiera que la matrice est dupliquée sur chaque site, et non répartie, ce qui peut être utile dans le cas de petites matrices qui pourraient donner des speed-up inférieurs à 1 pour certains algorithmes.

Un autre type de redistribution présente un intérêt certain: la circulation (ou rotation) des morceaux de l'agrégat sur un anneau virtuel constitué par les différents processeurs. A chaque étape, chaque processeur envoie son morceau d'agrégat à son voisin de droite, et reçoit celui de son voisin de gauche. Après p étapes, chaque processeur a pu accéder localement à toutes les données de l'agrégat réparti, qui se retrouve avec sa distribution initiale. Ceci sera utilisé par exemple pour la multiplication de deux matrices.

Notre démarche va ensuite consister à redéfinir[2] dans la classe matrice répartie d'abord les méthodes affectées par la répartition des données (création et accès aux données), puis celles dont on veut optimiser le code en fonction de cette distribution (addition, multiplication, inversion).

Grâce à l'encapsulation stricte, la seule façon d'accéder aux éléments d'une matrice est d'utiliser les méthodes:

- $item(i,j)$, qui retourne la valeur de l'élément présent à la ligne i, colonne j

[2]Si on utilise C++, cela implique que toutes les méthodes de la classe matrice soient déclarées "virtual". Ceci est implicite en Eiffel.

- *put*$(valeur, i, j)$, qui range une valeur en ligne i, colonne j.

Dans la classe matrice répartie, nous redéfinissons donc ces deux méthodes pour les adapter à notre distribution de données. Pour *Put*, on utilise le principe d'écriture locale (*Exec*): seul le processeur qui possède l'élément (i, j) peut en modifier la valeur, les autres n'effectuent aucune opération. Pour *Item*, on utilise l'instruction *Refresh* de la classe agrégat réparti qui permet la lecture distante d'une valeur, ainsi que différentes autres fonctionnalités de cette classe comme par exemple des fonctions de transformation d'index (qui permettent d'établir les correspondances entre index absolus de l'agrégat et couples numéros de processeur, index relatif à ce processeur de l'agrégat réparti). Notre classe matrice répartie devient donc:

```
Class Distributed_Matrix
    export repeat Matrix -- Same interface than Matrix
    inherit Distributed_Aggregate; Matrix -- inherit from both ancestors
            rename item as local_item, put as local_put;
            redefine item, put
feature
    item (row, column: integer): real is
        do Result := Refresh (row, column) end;
    put (v: like item; row, column: integer) is
        do if Exec (row, column)
             then local_put (v, absolute_to_local_index(row),column) end;
        end
end;
```

Nous avons maintenant une classe matrice répartie ayant la même sémantique (i.e. toutes les méthodes exportées ont la même sémantique) que la classe *Matrix*; mais pour l'instant, le seul résultat visible sera une perte d'efficacité lorsqu'on utilisera les opérations de *Matrice_répartie*, car les nouveaux mécanismes d'accès aux éléments de la matrice sont plus coûteux que les anciens. C'est pourquoi il nous faut redéfinir les opérations de la classe matrice répartie en réalisant les optimisations nécessaires pour obtenir le type de parallélisme mis en évidence plus haut.

Certaines de ces optimisations sont tout à fait classiques: on les trouve dans SUPERB, FORTRAN-D, PANDORE etc. L'originalité de notre démarche consiste ici à en étudier l'application dans un cadre de LAO, qui nous permettra en outre de présenter et de mettre en œuvre de nouvelles idées.

3.4 Optimisations du schéma de base

Suppression des Refresh/Exec inutiles Si dans dans un algorithme on sait qu'un élément de l'agrégat est présent localement, on peut y accéder directement en utilisant les méthodes "put" et "item" de la superclasse *Matrix*, ce qui économise des tests et évite de générer des communications inutiles. De la même manière, si on peut savoir statiquement qu'une donnée ne change pas entre

```
redefine infix "+";
infix "+" (m: like Current): like Current
    require
        m_must_exist: not m.Void;
        compatible: nb_rows=m.nb_rows and nb_columns=m.nb_columns
    local
        i,j : INTEGER;
    do
        result.Create(nb_rows,nb_columns);
        from i := 1 until i > local_nb_rows --Restricting iteration domain
        loop
            from j := 1 until j > nb_columns
            loop
                result.local_put(local_item(i,j) + m.local_item(i,j),i,j);
                j := j+1
            end;
            i := i+1
        end;
end; -- infix "+"
```

Figure 7 : Addition de deux matrices réparties avec EPEE

deux lectures distantes, on peut économiser une opération Refresh en utilisant une variable temporaire.

Restriction des domaines d'itérations Pour l'instant, lors d'une itération chaque processeur parcourt l'itération complètement, alors qu'il n'effectue que les calculs écrivant des données locales. Il est bien connu qu'en restreignant les domaines d'itération aux données locales de chaque processeur, on divise par p la taille du problème à résoudre, ce qui donne en théorie un speed up proportionnel. L'addition de deux matrices, présentée en figure 7, est un exemple d'utilisation de ces deux optimisations.

Optimisation des fonctions de transformation d'index Le projet Pandore a permis de mettre en évidence le coût prohibitif de ces fonctions de transformation d'index. Là encore, l'encapsulation va nous permettre d'apporter une solution élégante à ce problème, au prix d'une consommation supplémentaire de mémoire en $O(n)$.

En effet, il suffit de calculer une fois pour toutes ces correspondances d'index et de les stocker dans une table qui sera accédée directement par les fonctions de transformation d'index redéfinies de façon had hoc. Cette table devra être mise à jour à la création de la matrice et lors de chaque redistribution de données,

```
redefine infix "*";
infix "*" (m: like Current): like Current
    require
        m_must_exist: not m.Void;
        compatible: nb_columns=m.nb_rows
    local
        p,i,j,k: INTEGER;
    do
        Result.Create(nb_rows,m.nb_columns);

        from p := 1 until p > number_of_nodes loop
            from k := 1 until k > m.local_nb_rows loop
                from i := 1 until i > Result.local_nb_rows loop
                    from j := 1 until j > Result.nb_columns loop
                      Result.local_put(Result.local_item(i,j)
                        + local_item(i,m.local_to_absolute_index(mynode,r,k))
                        * m.local_item(k,j),
                        i,j);
                      j := j+1
                    end;
                    i := i+1
                end;
                k := k +1
            end;
            m.circulate(p); -- shift m data already processed to neighbor
            p := p + 1
        end;
end; -- infix "*"
```

Figure 8 : Multiplication de deux matrices réparties en Eiffel

mais elle est encapsulée dans la classe matrice répartie, donc invisible pour l'utilisateur. Ce calcul d'index ne sera donc plus pénalisé que d'une simple indirection supplémentaire.

Redistribution de l'agrégat en fonction des opérations Comme nous disposons d'une fonction permettant de redistribuer les données, nous pouvons faire en sorte que chaque opération effectue sur ses données les redistributions optimales pour l'algorithme utilisé, comme l'illustre en figure 8 l'utilisation de la possibilité de faire circuler les morceaux de l'agrégat réparti pour implanter la multiplication de deux matrices.

3.5 Implantation de EPEE

Notre Environnement Parallèle d'Exécution de Eiffel est constitué de trois parties :

- la classe Distributed Aggregate, qui est une classe Eiffel normale, faisant largement usage de fonctions C externes. Cette classe doit être héritée par chaque classe Eiffel voulant tirer partie des possibilités de parallélisme et de distribution existant dans EPEE
- un ensemble de modules d'interface destinés à fournir une interface homogène pour la classe Distributed Aggregate, et instrumentée pour en faciliter l'expérimentation. Ces modules sont écrits en C et bâtis au dessus de l'environnement d'expérimentation ECHIDNA [9]. Ils sont disponibles pour réseaux de Suns, hypercubes Intel iPSC/2 et iPSC/860.
- un ensemble d'outils pour faciliter la compilation croisée et l'expérimentation distribuée de programmes Eiffel avec EPEE.

Nous avons mis en œuvre ces idées à l'aide du compilateur Eiffel 2.3 de ISE, qui offre la possibilité de générer des paquetages C autonomes que nous avons pu compiler sans trop de problèmes pour des environnements spécialisés.

Notons quelques particularités de notre implantation par rapport à la plupart des implantations de langages à objets parallèles.

1. Nous compilons un programme purement Eiffel, avec la sémantique séquentielle de Eiffel, en utilisant le compilateur original sans aucune modification. La seule différence est une augmentation des performances. Nous avons exprimé le concept de programmation SPMD sans toucher au langage, simplement à l'aide de la classe *Agrégat_réparti*.
2. Par rapport aux langages à objets MIMD, nos migrations d'objets sont encapsulées dans des classes, ce qui permet de s'affranchir d'un système de nommage

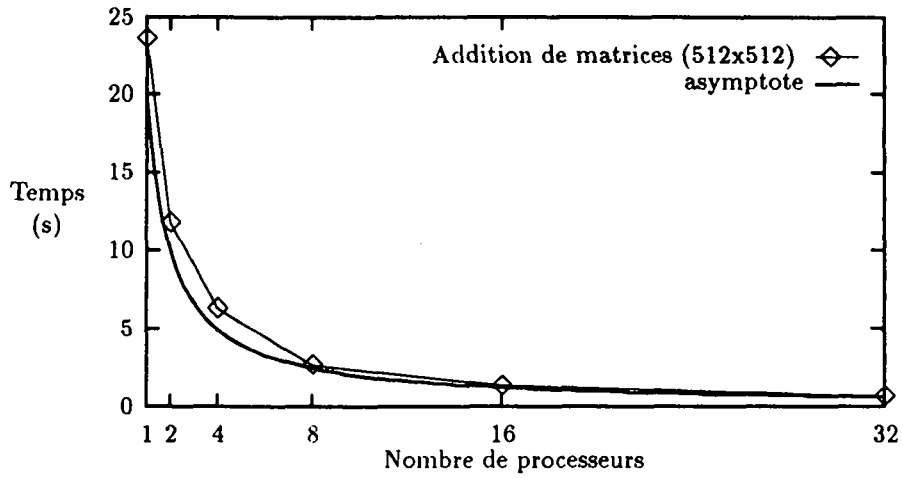

Figure 9 : Addition de matrices : performances

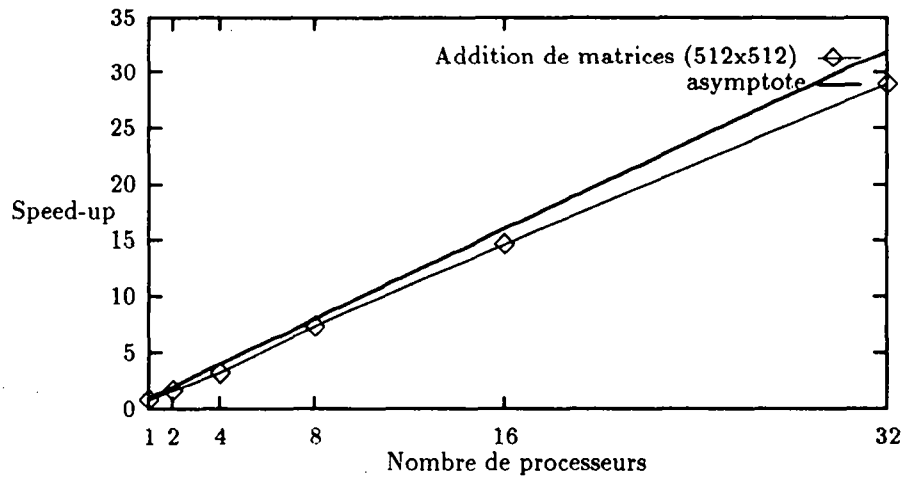

Figure 10 : Addition de matrices : speed-up

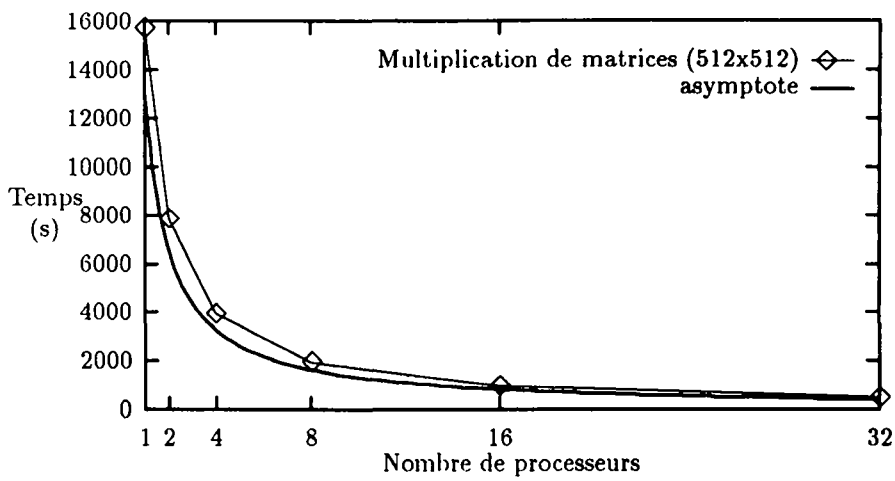

Figure 11 : Multiplication de matrices : performances

global pour les objets, extrèmement coûteux en communications et synchronisations.

3. Tous les objets ont une composante locale sur chaque site, qu'ils soient dupliqués (par défaut) ou morcelés (s'ils héritent de la classe agrégat réparti). Donc l'algorithme de ramasse-miettes séquentiel de Eiffel nous suffit (on peut même le faire tourner en parallèle avec un algorithme bloqué sur attente de message), et évite l'implantation d'un coûteux algorithme de ramasse-miettes distribué.

4. Débogage réparti : il est possible d'utiliser l'environnement Eiffel dans plusieurs processus Unix —représentant chacun un processeur— sur une seule station de travail, afin de mettre au point, algorithme par algorithme, la classe qu'on veut morceler (ici *Distributed_Matrix*). Notre modèle de parallélisme étant entièrement encapsulé dans la classe agrégat réparti, et conçu pour être totalement extensible, nous avons la garantie que notre programme tournera sans modification et de la même manière quelle que soit la machine sous-jacente, ce qui a été confirmé par nos expériences.

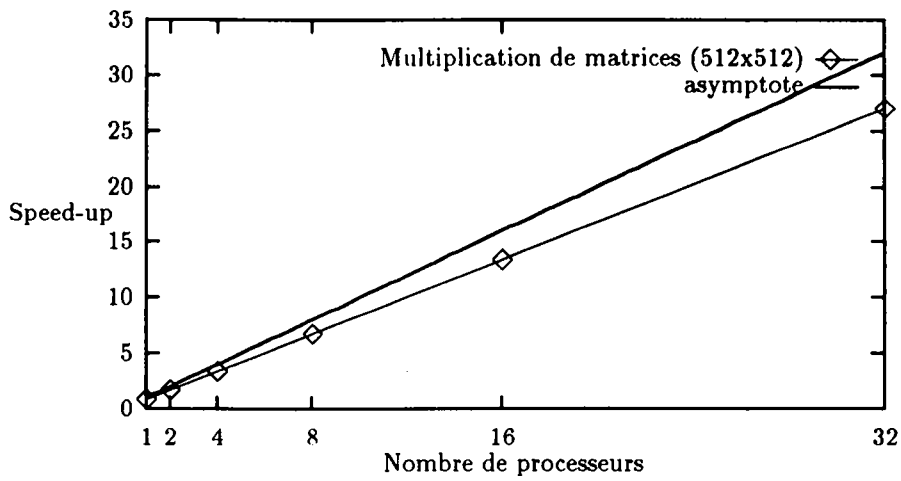

Figure 12: Multiplication de matrices: speed-up

3.6 Mesures de performances et commentaires

Nous avons conduit quelques campagnes de mesures de performances de notre implantation sur un hypercube Intel iPSC/2 muni de 32 processeurs (80386). Nous avons mesuré pour différentes tailles de cubes les temps d'exécution par processeur des méthodes +, × et *Inversion* (voir figures 9,11,et 13). Nous les avons comparées aux meilleures versions séquentielles de ces mêmes méthodes exécutées sur un nœud de l'hypercube, ce qui permet de calculer le speed-up (voir figures 10,12,et 14).

Remarquons que dans tous les cas les speed-up sont quasi-linéaires en fonction du nombre de processeurs, ce qui confirme l'intérêt de notre approche pour l'aspect extensibilité du parallélisme en fonction du nombre de processeurs de l'APMD.

La différence de performances entre l'exécution sur un seul processeur d'une méthode de la classe répartie et l'exécution de sa version séquentielle est sans doute due au surcoût de la liaison dynamique des méthodes "put" et "item", et dans une moindre mesure à la double indirection de calcul des index. C'est cette différence qui conditionne l'écart entre les courbes réelles et les asymptotes. Aussi un mécanisme de type "cache" tel que réalisé dans Sather (dialecte efficace de Eiffel) pour diminuer le coût de la liaison dynamique permettrait sans doute d'améliorer cet état de fait.

Notons finalement que la vitesse absolue (en nombre d'opérations flottantes efficaces par seconde) que nous obtenons est d'environ 600kFlops sur 32 processeurs (pour la multiplication de matrices), alors qu'une implantation directe en *C* atteint

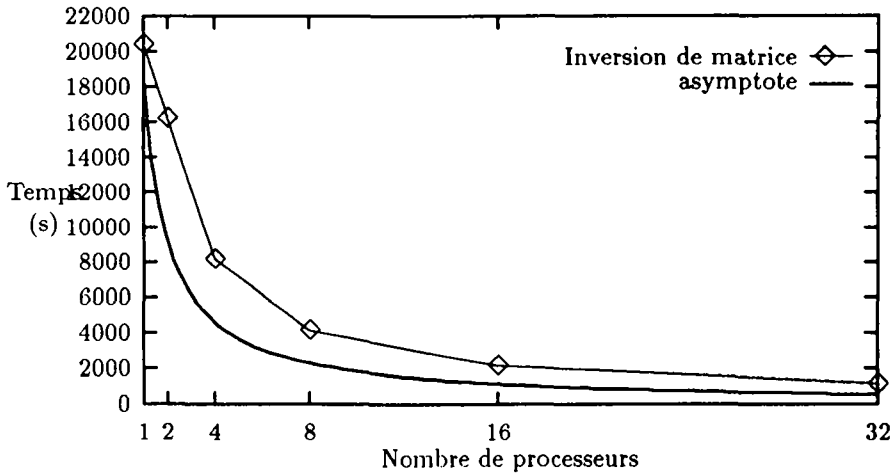

Figure 13 : Inversion de Matrices : performances

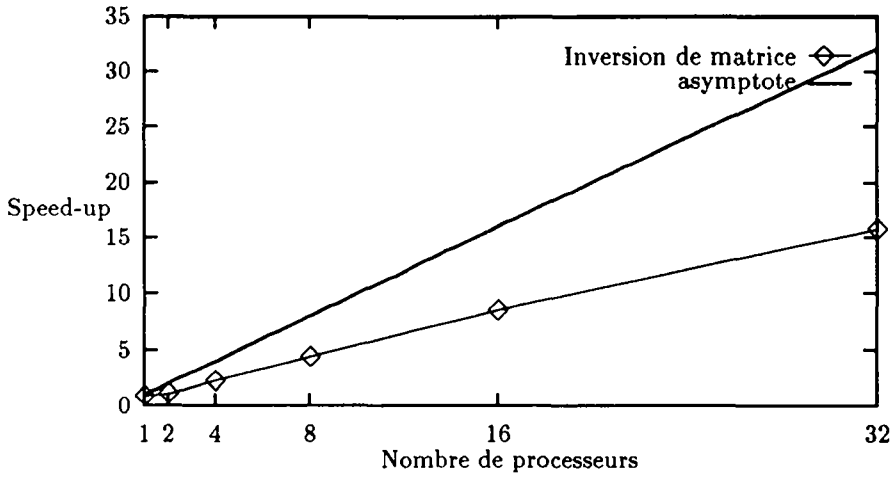

Figure 14 : Inversion de Matrices : speed-up

les 3MFlops, i.e. est 5 fois plus rapide. Comme le montrent les tests séquentiels Eiffel/*C* tant sur l'hypercube que sur d'autres machines, ceci semble être lié à l'efficacité peu satisfaisante du code généré par la version 2.3.4 du compilateur Eiffel de ISE. Ceci devrait s'améliorer avec les futures versions de compilateurs Eiffel disponibles en 1992 (ISE Version 3, SIG, Sather...).

4 Conclusion

Nous avons montré l'intérêt de notre approche pour faciliter la programmation d'APMD : nous avons présenté le modèle de programmation séquentiel pour le programmeur d'application (qui ne voit l'APMD que comme un processeur de calcul plus puissant) et un modèle SPMD pour le concepteur de classes réparties. Bien sûr notre modèle n'est pas universel, mais au contraire d'autres types d'approches, il s'adapte bien à l'expression d'un parallélisme massif pourvu que les problèmes à résoudre soient de taille assez grande (mais sinon pourquoi faire appel à des supercalculateurs?). Ce modèle de programmation présente en outre l'avantage de permettre la récupération (réutilisation des librairies de classes) des environnements de programmation séquentiels déjà existants, bien sûr sans gain de performance, mais sans perte non plus, ce qui ne serait pas le cas dans un modèle où la plupart des objets seraient des entités actives.

Au niveau méthodologique, nous avons mis en évidence le fait que les constructions et mécanismes classiques des langages à objets suffisent pour exprimer le modèle SPMD : nul besoin d'altérer le langage. Notamment, l'héritage multiple nous a permis d'utiliser et d'exprimer clairement le concept unificateur d'*agrégat réparti*. La modularité et l'encapsulation nous ont permis d'assurer la correction sémantique des accès aux données de l' agrégat réparti, et ceci de manière transparente. Le polymorphisme (liaison dynamique) nous a permis de réaliser les optimisations nécessaires à l'obtention du speed-up souhaité.

Cependant, comme nous l'avons vu, les résultats de performances que nous obtenons sont intéressants mais pas encore totalement satisfaisants. En dehors des points liés à l'utilisation d'un langage à objet en général (coût de la liaison dynamique) et de Eiffel en particulier (implantation peu efficace), les principales limitations proviennent de l'utilisation du modèle SPMD quand toutes les optimisations ne sont pas mises en œuvre. Une autre limitation concerne le problème non trivial du traitement et de la propagation des exceptions non fatales, de manière à ce que l'exécution répartie soit absoluement identique à l'exécution séquentielle.

Au delà de la résolution de ces problèmes, EPEE ouvre plusieurs perspectives de recherche. En effet, l'approche orientée objet est un bon moyen —si ce n'est le seul— d'affronter le problème de la parallélisation d'opérations sur des structures de données complexes comme des ensembles, des arbres, des graphes, etc. De plus, la facilité et la souplesse d'utilisation des LAO nous permettent de réaliser des études

pratiques sur des problèmes concrets de parallélisation à l'aide d'expérimentations sur différentes APMD, afin d'obtenir des informations plus précises sur le comportement de programmes obtenus par ce type de techniques, ce qui est un des thèmes de recherche de l'équipe PAMPA de l'IRISA.

D'autre part, la tâche de parallélisation effective d'une classe nécessite encore une grande expertise à la fois du domaine d'application et des techniques de parallélisation. Or la méthode que nous proposons semble être assez générale et systématique pour pouvoir se prêter à une certaine forme d'automatisation, peut-être par exemple sous la forme d'outil intéractif d'aide à la parallélisation.

Finalement, il est intéressant de remarquer que cette approche générale consistant à encapsuler des détails d'architecture dans des classes *had hoc* n'est pas limitée au contexte de la programmation d'APMD : elle pourrait être aussi utilisée avec des co-processeurs vectoriels ou systoliques, fournissant ainsi une alternative possible à la solution classique consistant à utiliser des compilateurs dédiés.

Références

[1] Pierre America. *POOL-T: a Parallel Object-Oriented Language*, pages 200–220. MIT Press, 1987.

[2] Birger Andersen. Ellie language definition report. *Sigplan Notices*, 1990.

[3] Françoise André, Jean-Louis Pazat, and Henry Thomas. Pandore: A system to manage Data Distribution. In *International Conference on Supercomputing*, ACM, June 11-15 1990.

[4] John K. Bennett. The design and implementation of DistributedSmalltalk. In *OOPSLA'87 Proceedings*, 1987.

[5] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[6] Rohit Chandra, Anoop Gupta, and John L Hennessy. *COOL: a Language for Parallel Programming*, chapter 8. Gelernter, D. et al., 1990.

[7] C. M. Chase, A. L. Cheung, A. P. Reeves, and M. R. Smith. Paragon: a parallel programming environment for scientific applications using communication structures. In *1991 International Conference on Parallel Processing*, 1991.

[8] A. A. Chien and W. J. Dally. Concurrent aggregates (ca). In *Proc. of the Second ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, 1991.

[9] C. Jard and J.-M. Jézéquel. A multi-processor Estelle to *C* compiler to experiment distributed algorithms on parallel machines. In *Proc. of the* 9^{th} *IFIP International Workshop on Protocol Specification, Testing and Verification, University of Twente, The Netherlands*, North Holland, 1989.

[10] Michael F. Kilian. Object-oriented programming for massively parallel machines. In *1991 International Conference on Parallel Processing*, 1991.

[11] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[12] Leslie G. Valiant. A bridging model for parallel computation. *CACM*, 33(8), Aug 1990.

[13] Youfeng Wu. Parallelism encapsulation in C++. In *1990 International Conference on Parallel Processing*, 1990.

[14] Yasuhiko Yokote and Mario Tokoro. The design and implementation of ConcurrentSmalltalk. In *OOPSLA'86 Proceedings*, 1986.

[15] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA'86 Proceedings*, 1986.

[16] Hans P. Zima, Heinz-J. Bast, and Michael Gerndt. SUPERB: a tool for semi-automatic MIMD /SIMD parallelization. *Parallel Computing*, (6):1-18, 1988.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 616 USING COHERENCE TO ACCELERATE RADIOSITY
Pierre TELLIER, Eric MAISEL, Kadi BOUATOUCH, Eric LANGUENOU
Novembre 1991, 16 pages.

PI 617 INTERVAL APPROXIMATIONS OF MESSAGE CAUSALITY IN DISTRIBUTED EXECUTION
Claire DIEHL, Claude JARD
Novembre 1991, 44 pages.

PI 618 RETOUR SUR LE RESEAU SYSTOLIQUE DU PALINDROME
Hervé LE VERGE, Patrice QUINTON
Novembre 1991, 14 pages.

PI 619 TRANSFORMATIONS DU GRAPHE DES PROGRAMMES SIGNAL
Olivier MAFFEIS, Bruno CHERON, Paul LE GUERNIC
Novembre 1991, 82 pages.

PI 620 METHODES D'ANALYSE EVOLUTIVE EN TRAITEMENT D'ANTENNES
Olivier ZUGMEYER, Jean-Pierre LE CADRE
Décembre 1991, pages.

PI 621 GENERATING MEMORY-EFFICIENT IMPERATIVE DATA STRUCTURES FROM SYSTOLIC PROGRAMS
Zbigniew CHAMSKI
Décembre 1991, 20 pages.

PI 622 ELEMENTS FOR A COURSE ON THE DESIGN OF DISTRIBUTED ALGORITHMS
Noël PLOUZEAU, Michel RAYNAL
Décembre 1991, 12 pages.

PI 623 PARALLELISME MASSIF ET LANGAGE A OBJETS : UNE APPROCHE SPMD
Jean-Marc JEZEQUEL
Décembre 1991, 26 pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399