Laboratoire d'Informatique Fondamentale de Lille

Numéro d'ordre: 2561

THÈSE

Présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

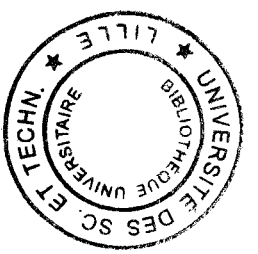
Pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Guy BERGÈRE

CONTRIBUTION À UNE PROGRAMMATION PARALLÈLE HÉTÉROGÈNE DE MÉTHODES NUMÉRIQUES HYBRIDES



Thèse soutenue le 22 septembre 1999, devant la Commission d'Examen:

Président:	Jean-Luc DEKEYSER,	Université de Lille I
Rapporteurs:	Paul FEAUTRIER,	Université de Versailles Saint-Quentin-en-Yvelines
	Bertil FOLLIOT,	Université de Paris VI
Examinateurs:	Michel COSNARD,	ENS Lyon - INRIA Lorraine
	Pierre MANNEBACK,	Faculté polytechnique de Mons
	Guy-René PERRIN,	Université de Strasbourg I
	Serge PETITON,	Université de Lille I
	El-Ghazali TALBI,	Université de Lille I

À mon grand-père,
autodidacte s'il en fut ...

Remerciements

Je remercie tout d'abord M. Serge PETITON, Professeur à l'Université des Sciences et Technologies de Lille, pour m'avoir accueilli dans son équipe et avoir accepté de diriger ma thèse.

Je remercie M. Paul FEAUTRIER, Professeur à l'Université de Versailles Saint-Quentin-en-Yvelines, ainsi que M. Bertil FOLLIOT, Professeur à l'Université de Paris VI, d'avoir accepté de rapporter ma thèse et pour les échanges constructifs en ayant résulté.

Je tiens à remercier M. Jean-Luc DEKEYSER, Professeur à l'Université des Sciences et Technologies de Lille, pour les échanges fructueux avec l'équipe qu'il dirige et pour avoir accepté de présider le jury de cette thèse.

Je remercie M. El-Ghazali TALBI, Maître de Conférence à l'Université des Sciences et Technologies de Lille, pour les échanges constructifs concernant le système MARS et pour avoir accepter d'être membre du jury.

Je remercie également M. Michel COSNARD, Professeur à l'ENS de Lyon et à l'INRIA de Lorraine, M. Pierre MANNEBACK, Professeur à la Faculté Polytechnique de Mons, et M. Guy-René PERRIN, Professeur à l'Université de Strasbourg, pour avoir accepté d'être membres du jury de cette thèse.

Je tiens particulièrement à remercier M. Azeddine ESSAI, du laboratoire d'Analyse Numérique et d'Optimisation, pour sa collaboration dans l'étude de la méthode hybride parallèle et pour ses nombreux éclairages concernant les méthodes numériques. Je le remercie en particulier pour sa disponibilité sans faille et sa gentillesse.

Je remercie aussi M. Djamai KEBBAL, dont les conseils m'ont été précieux dans l'utilisation de MARS, et pour les nombreuses discussions constructives dans ce domaine.

Je remercie l'Établissement Technique Central de l'Armement d'avoir mis à notre disposition, Azzedine et moi, la machine CM5 et plusieurs stations SUN pour l'expérimentation de la méthode hybride entrelacée.

Je remercie également tous les membres de l'équipe WEST dont j'ai partagé un temps le bureau et qui m'ont accueilli avec gentillesse, tout en me faisant profiter de leur riche expérience du parallélisme.

Enfin je remercie M. Bernard PERON, proviseur du lycée Maurice Duhamel à Loos où je suis enseignant, de m'avoir permis de disposer d'un emploi du temps compatible avec mes études doctorales, et grâce auquel cette thèse a pu voir le jour.

Résumé

De multiples applications scientifiques ou industrielles nécessitent la résolution de systèmes linéaires non symétriques, décrits par des matrices creuses de très grande taille. De tels problèmes impliquent souvent l'usage de méthodes numériques itératives, ainsi que le recours au parallélisme. La méthode GMRES(m) donne de bons résultats mais sa parallélisation est limitée par les nombreuses communications engendrées. Sa convergence peut être accélérée grâce à la connaissance des valeurs propres, qui peuvent être calculées séparément sur une autre machine.

Ceci conduit à la méthode hybride, décrite dans la première partie de cette thèse. Sa mise en œuvre a été élaborée en collaboration avec Azeddine Essai, du « laboratoire d'Analyse Numérique et d'Optimisation », chargé des aspects numériques du problème. Deux versions ont été réalisées : la première utilise deux machines parallèles très différentes et elle peut être qualifiée de « parallèle hétérogène asynchrone », l'autre n'utilise qu'une machine parallèle en tirant parti des délais d'attente induits par les calculs séquentiels déportés, et elle est donc « parallèle entrelacée ».

Dans la seconde partie, il s'agit de tirer profit de l'expérience acquise et des questions soulevées lors de la mise au point de ces implantations. Un objectif à court terme est l'écriture des différentes parties composant cette méthode numérique sous forme de composants logiciels génériques, pouvant se paralléliser de différentes façons, afin de pouvoir mettre en œuvre cette méthode dans d'autres environnements parallèles, hétérogènes ou non. Un objectif plus ambitieux est de définir les spécifications d'un langage de contrôle, permettant l'écriture aisée d'applications numériques hybrides pour des environnements parallèles éventuellement hétérogènes. Le système MARS, qui permet d'obtenir une granularité adaptative en fonction de la charge des machines, a servi de base à la spécification de ce langage de contrôle.

Mots clefs : matrices creuses, méthodes itératives, méthodes hybrides, parallélisme hétérogène, parallélisme asynchrone, composants logiciels, langages de contrôle.

Abstract

Several scientific or industrial applications require to solve non symmetric linear systems described with very large sized sparse matrices. Such problems implie often to use numerical iterative methods, and to resort parallelism. GMRES(m) method give good results, but its parallelization is limited with numerous generated data communications. Its convergence may be accelerated with eigenvalues knowledge. These values may be separately computed on another machine.

That leads to the hybrid method, described in the first part of this thesis. Its achievement has been developed in collaboration with Azeddine Essai, from "laboratoire d'Analyse Numérique et d'Optimisation", in charge of numerical points of view of our problem. Two versions has been achieved: the first one uses two parallel very different machines and may be called "parallel heterogeneous asynchronous", the other one uses only one parallel machine by taking advantage of waiting delays inferred by deported sequential computations, and it is called therefore "interleaved parallel".

In the second part, it concerns taking advantage of acquired experience and questions raised during the development of these implementations. A short-range aim is to write the different parts of this numerical method as generic software components, able to be parallelized with different manners, in order to be able to implement this method in others parallel environments, heterogeneous or not. A more ambitious aim is to define specifications of a control language allowing to write easily numerical hybrid applications for parallel environments, possible heterogeneous. The MARS system, which allows to obtain an adaptative granularity in accordance with machines load, has been used to base specifications of this control language.

Keywords: sparse matrices, iterative methods, hybrid methods, heterogeneous parallelism, asynchronous parallelism, software components, control languages.

Table des matières

Chapitre 1

Introduction

La résolution de systèmes linéaires est une partie importante de nombreux problèmes scientifiques ou industriels. Ces systèmes sont souvent exprimés sous la forme de matrices non symétriques, creuses et de très grande taille. La résolution de tels systèmes a fait l'objet de nombreuses recherches, aussi bien à propos des méthodes numériques employées qu'au sujet de leur mise en œuvre informatique. La taille des problèmes concernés et leur coût en temps de calcul incite à optimiser les méthodes employées, et impose l'usage du parallélisme.

Le travail présenté ici à ce propos a été réalisé au sein d'une équipe pluridisciplinaire, née de la collaboration entre le laboratoire d'Analyse Numérique et d'Optimisation (ANO) et le Laboratoire d'Informatique Fondamentale de Lille (LIFL).

Dans le cadre de cette collaboration, une méthode numérique itérative hybride a été mise au point, afin de tenter d'atteindre les objectifs suivants :

- Obtenir une convergence numérique en un moins grand nombre d'itérations ;
- Dépasser la limitation du degré de parallélisme due à une granularité handicapée par le volume des communications, en utilisant autrement le parallélisme disponible ;
- Augmenter le parallélisme disponible par le recours à l'hétérogénéité, permettant d'utiliser en appoint des stations de travail ou des machines parallèles.

Si le premier problème est du ressort de l'analyse numérique, le second et le troisième posent la question plus vaste de toute application développée en parallélisme hétérogène.

En premier lieu, c'est le découpage même de l'application en différents composants logiciels qui est au cœur du problème. Savoir quelle partie tirera mieux avantage de quelle architecture séquentielle ou parallèle est fondamental.

L'articulation entre ces différentes parties est aussi très importante. Le volume des communications entre deux composants ainsi que la nature des synchronisations sont des

éléments influant considérablement sur les performances.

Mais le modèle théorique établi pour une application donnée doit souvent être adapté en fonction de la disponibilité, des caractéristiques et des possibilités des machines réellement utilisables. Et inversement, une application bâtie efficacement autour du matériel présent sur un site perdra de son intérêt si sa portabilité est difficile. Il convient donc à la fois d'établir un modèle abstrait de parallélisation hétérogène idéale et de se donner les moyens d'implantations pragmatiques tenant compte des matériels disponibles et de leurs caractéristiques.

Compte tenu de la complexité d'une telle parallélisation hétérogène, d'autant plus importante qu'elle ne concerne généralement que des applications d'envergure, le travail du programmeur peut être considérablement allégé si certains de ces aspects sont pris en compte par un langage de contrôle. Celui-ci peut notamment lui permettre de faire la distinction entre les tâches de base, pouvant éventuellement être puisées dans des bibliothèques, et l'articulation entre ces tâches, pour laquelle il pourra disposer d'un certain nombre d'abstractions et d'outils tels que :

- la gestion des différentes tâches ;
- l'accès à la mémoire distante et les communications ;
- le respect d'un arbre de précédence et les synchronisations ;
- la prise en compte de la charge des machines.

Cette thèse développe dans un premier temps la mise au point de la méthode hybride, à partir de laquelle les questions qui viennent d'être exposées ont été mises en évidence. Les méthodes numériques employées, décrites dans la *section 2.2*, sont des méthodes itératives, et la particularité de l'approche du problème, au point de vue numérique, réside dans l'hybridation des méthodes « GMRES » pour la résolution du système, « Arnoldi » pour le calcul des valeurs propres et « Least Squares » pour la prise en compte de celles-ci dans le calcul d'un nouvel itéré permettant l'accélération de la convergence. Cette méthode numérique hybride et ses aspects théoriques du point de vue de l'analyse numérique font l'objet de la thèse d'Azeddine Essai : « Méthode hybride parallèle hétérogène et méthodes pondérées pour la résolution des systèmes linéaires » [13].

L'aspect informatique du problème est lui aussi inhabituel, car il exploite à la fois le parallélisme, l'hétérogénéité et l'asynchronisme. En effet, comme cela sera montré dans la *section 2.2.1*, le degré de parallélisation autorisé par les méthodes numériques telles que GMRES et Arnoldi est limité par le grand nombre des communications induites, et l'amélioration des performances doit être recherchée par d'autres voies que le simple accroissement du nombre des processeurs.

L'hybridation de plusieurs méthodes numériques est un cas idéal permettant d'exploiter davantage de parallélisme potentiel en assignant les calculs de chaque méthode à une machine parallèle différente, ou à une partition distincte de processeurs d'une même machine. L'absence de dépendances de données entre les différents algorithmes numériques de la méthode hybride (les échanges de données n'opérant qu'un enrichissement en vue d'accélérer la convergence), autorise un asynchronisme total et offre les meilleures conditions pour utiliser l'hétérogénéité d'un réseau au sein d'une application. Cet aspect permet de profiter au mieux de l'ensemble des machines parallèles accessibles sur un réseau donné, y compris de celles qui sont sous-exploitées en raison de leurs performances un peu dépassées.

Deux implantations de la méthode hybride précitée ont été réalisées :

- une version asynchrone hétérogène, utilisant deux machines parallèles et quelques machines séquentielles, est décrite au *chapitre 4*,
- une version entrelacée, profitant des délais d'attente d'une seule machine parallèle, à propos des calculs séquentiels déportés sur des machines plus appropriées à ce type de calcul, est décrite au *chapitre 5*.

Dans les deux cas, les résultats obtenus montrent l'intérêt à la fois de la méthode numérique et des choix concernant sa mise en œuvre informatique. Deux problèmes cependant, liés à tout calcul parallèle sur les matrices creuses, n'ont pas été abordés dans cette thèse car ils auraient nécessité des développements trop importants :

- L'optimisation du placement des données, qui n'est pas un problème simple notamment pour obtenir une parallélisation optimale des communications avec une architecture massivement parallèle [37, 50].
- Le cas des matrices ne pouvant être logées entièrement en mémoire. Un traitement par bloc est alors inévitable ce qui implique un partitionnement de la matrice. L'optimisation de celui-ci est un problème NP-complet [32].

Les machines évoluant constamment, et les sites susceptibles d'accueillir une version d'une application hétérogène conséquente étant très divers, l'adaptabilité de ces implantations est une qualité essentielle, ainsi que la modularité des composants qui réutilisent souvent des algorithmes de calcul bien connus, pouvant être tirés sans réécriture de bibliothèques logicielles existantes. Il s'agit aussi de tirer le meilleur parti possible de parcs de machines séquentielles et parallèles de performances et d'architectures très diverses, qui peuvent être couplées au sein d'un réseau hétérogène, et se compléter judicieusement dans la mise en œuvre d'une application lourde telle que celle de la méthode hybride précitée. En conséquence, diverses options d'implantation possibles de cette méthode hybride, en plus de celles déjà mises en œuvre, sont étudiés au début du *chapitre 6*, en tenant compte de ces nécessités d'adaptabilité et de modularité, ainsi que des diverses architectures parallèles et des configurations hétérogènes existantes.

La mise au point de ce type d'application pose le problème plus général de la gestion des applications parallèles hétérogènes lourdes, et du bénéfice pouvant être retiré de l'exploitation de cette hétérogénéité. La lourdeur d'une écriture « à la main », gérant processus et communications à l'aide de primitives de bas niveau, paraît évidente. Le *chapitre 3* expose un certain nombre d'outils logiciels utilisables pour l'écriture de telles applications. Il révèle aussi que s'il existe de nombreux environnements de programmation parallèle, assez peu permettent de gérer efficacement l'hétérogénéité.

La structure d'une application hétérogène comprenant plusieurs modules parallèles, et les conditions d'une implantation efficace d'une telle application, sont décrites dans les *sections 6.4 à 6.7*. La mise à profit de l'asynchronisme entre certains modules y est notamment soulignée, ainsi que sa conséquence qui est le non-déterminisme du déroulement des calculs.

L'utilisation de l'environnement « MARS », qui permet une granularité adaptative du parallélisme, en fonction de la charge de chaque nœud de calcul, est abordée au *chapitre 7*. En se basant sur cet environnement, il est possible d'élaborer un langage de contrôle adapté aux applications parallèles hétérogènes, notamment à celles qui ont des propriétés asynchrones telles que les méthodes numériques hybrides étudiées précédemment. La disponibilité d'un tel langage de contrôle pourrait faciliter la mise au point d'applications parallèles hétérogènes toujours délicates à mettre en œuvre, en particulier à propos de la gestion des processus et de la granularité du parallélisme, ainsi que pour résoudre les problèmes de synchronisation et de communication. Sa spécification, non exhaustive, est formulée dans ce dernier chapitre.

L'abondance des besoins en terme de calcul numérique parallèle, et le développement exponentiel des réseaux hétérogènes pouvant accueillir des applications de ce type laissent entrevoir l'importance des besoins de recherche en ce domaine. Les perspectives évoquées dans la conclusion du présent travail n'en sont qu'un pâle reflet.

Chapitre 2

Résolution de systèmes linéaires non symétriques de très grande taille

De multiples applications scientifiques ou industrielles nécessitent la résolution de systèmes linéaires non symétriques, de très grande taille. Cette résolution requiert généralement des temps de calcul prohibitifs, sans commune mesure avec ceux nécessités par le traitement du reste du problème. L'optimisation de ce type de calcul est donc primordiale, et la grande taille de la structure de données implique le recours au parallélisme et à des méthodes appropriées.

Les systèmes linéaires en question sont en général exprimés sous la forme de matrices, la plupart du temps très creuses, de la spécificité desquelles il convient de tenir compte aussi bien pour la représentation en mémoire que pour le choix des méthodes de calcul.

2.1 Matrices creuses

2.1.1 Intérêt

Beaucoup de matrices de grande taille issues de problèmes industriels contiennent, un très grand nombre d'éléments nuls, et on parle alors de matrices « creuses ». La prise en compte du creux dans l'algorithme est très importante en terme d'efficacité. En effet, il est inutile d'affecter une place en mémoire aux éléments nuls, de même qu'il est inutile d'effectuer les calculs les concernant. On peut donc espérer gagner de la place en mémoire et du temps de calcul en gérant correctement le creux.

Il est donc nécessaire de compresser les matrices creuses, c'est à dire trouver un format de représentation de ces matrices [47, 41] qui permette :

- de ne pas mémoriser les éléments nuls,

- de retrouver facilement un élément connaissant ses coordonnées.

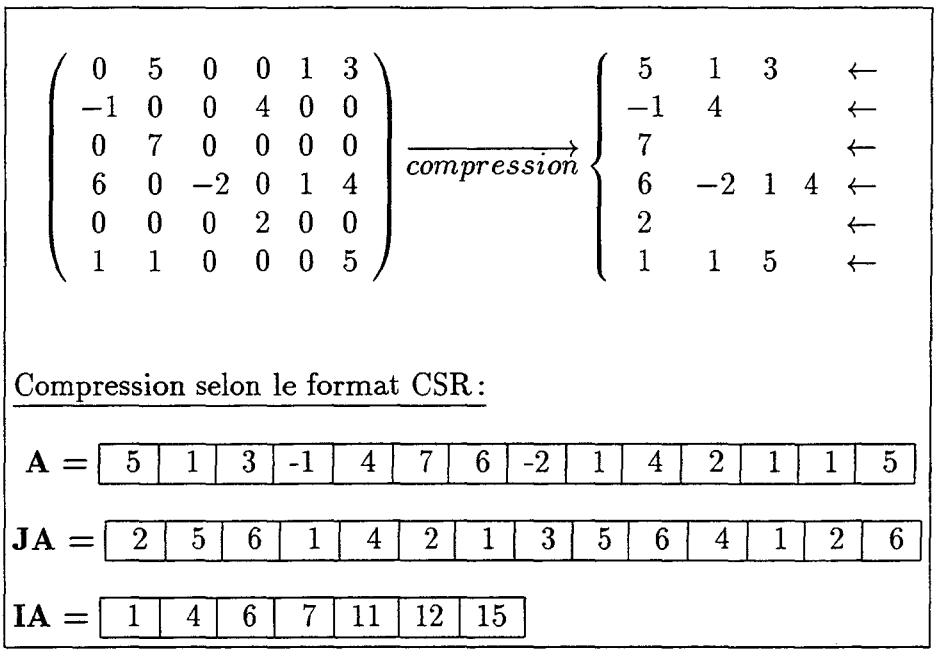


FIG. 2.1 - *Exemple de compression d'une matrice creuse au format CSR (compression et concaténation des lignes)*

- d'effectuer facilement les opérations courantes sur les matrices (notamment le calcul de la transposée).

2.1.2 Formats

Format CSR

Le format le plus économe en mémoire est le **CSR** (Compress Sparse Row) ou son homologue par colonnes CSC (Compress Sparse Column). Le CSR décrit la matrice creuse par trois vecteurs *(voir l'exemple de la figure 2.1)* :

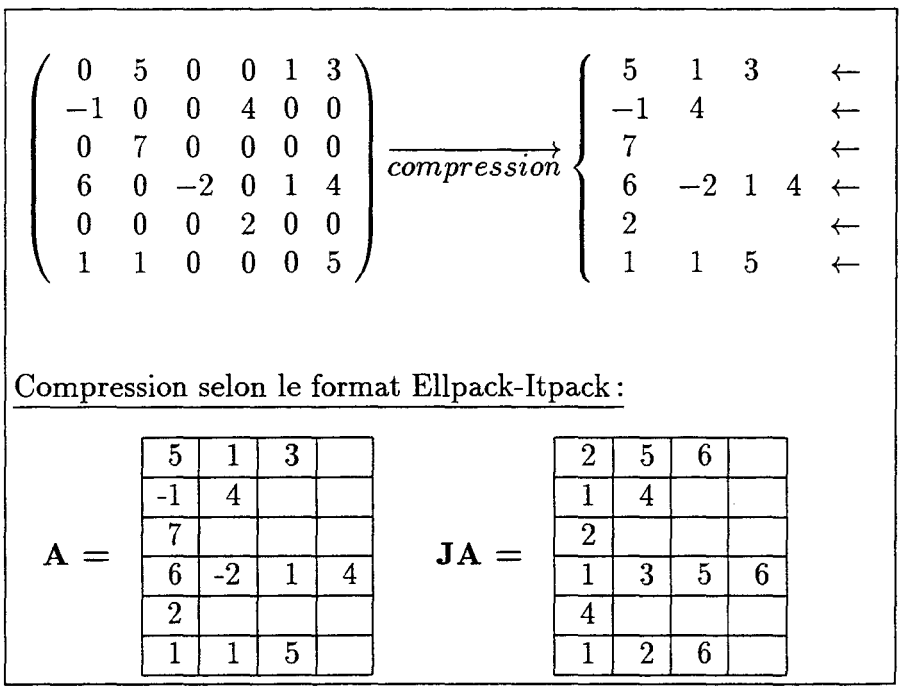
A est un vecteur contenant les éléments non nuls de la matrice, rangés ligne par ligne,

JA est un vecteur de même taille que A dont chaque élément contient le numéro de colonne de l'élément correspondant de A.

IA est un vecteur de taille $n+1$, où n est le nombre de lignes de la matrice. Chaque élément IA(i) de ce vecteur ($i \in [1, n]$) contient le rang, dans le vecteur A, du 1er élément de la ligne n° i de la matrice.
IA($n+1$) contient T+1, où T est la taille de A (et de JA).

L'accès à l'élément M(l, c) de la matrice M se fera donc de la façon suivante :

- Recherche des indices de début et de fin de la ligne n° l : $\begin{cases} début = IA(l) \\ fin = IA(l+1) - 1 \end{cases}$



0	5	0	0	1	3	5	1	3	←
−1	0	0	4	0	0	−1	4		←
0	7	0	0	0	0	7			←
6	0	−2	0	1	4	6	−2	1	4 ←
0	0	0	2	0	0	2			←
1	1	0	0	0	5	1	1	5	←

5	1	3	
-1	4		
7			
6	-2	1	4
2			
1	1	5	

2	5	6	
1	4		
2			
1	3	5	6
4			
1	2	6	

FIG. 2.2 - *Exemple de compression d'une matrice creuse au format Ellpack-Itpack (compression des lignes)*

- Recherche de la valeur c dans JA entre JA(*début*) et JA(*fin*) (Cette recherche peut être obtenue en complexité logarithmique).
- Si c est trouvé à la position JA(i), l'élément cherché est A(i); si c n'est pas trouvé, l'élément cherché est 0.

Format Ellpack-Itpack

Le format Ellpack-Itpack procède aussi à une compression des lignes. Il utilise deux tableaux de dimensions (1:n, 1:LMAX), où n est le nombre de lignes de la matrice d'origine, et LMAX le nombre maximum d'éléments non nuls dans une ligne de cette matrice *(voir l'exemple de la figure 2.2)* :

A est un tableau dont chaque ligne contient les éléments non nuls d'une ligne de la matrice,

JA est un tableau dont chaque élément contient le numéro de colonne de l'élément correspondant de A.

L'accès à l'élément M(l, c) de la matrice M, légèrement plus rapide que dans le format précédent, se fera de la façon suivante :

- Recherche de la valeur c dans la ligne JA(l) (Cette recherche peut être obtenue en complexité logarithmique).

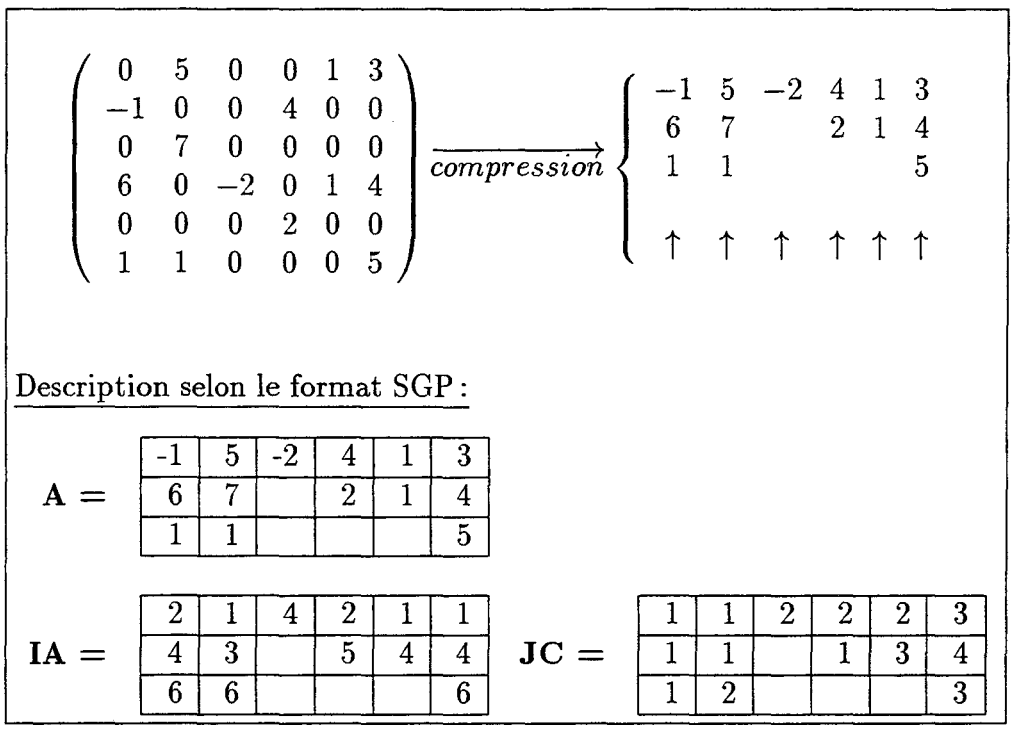


FIG. 2.3 - *Exemple de compression d'une matrice creuse au format SGP (compression des colonnes)*

- Si c est trouvé à la position JA(i), l'élément cherché est A(i); si c n'est pas trouvé, l'élément cherché est 0.

Ce format, plus gourmand en mémoire que le précédent (car certaines lignes contiennent moins de LMAX éléments non nuls), est intéressant car il permet un découpage facile de la matrice entre plusieurs processus parallèles. Il est particulièrement bien adapté à une parallélisation SPMD à gros grain.

Format SGP

Le format SGP fait une compression des colonnes. Il utilise trois tableaux de dimensions (1:n, 1:CMAX), où n est le nombre de colonnes de la matrice d'origine, et CMAX le nombre maximum d'éléments non nuls dans une colonne de cette matrice *(voir l'exemple de la figure 2.3)* :

A est un tableau dont chaque colonne contient les éléments non nuls d'une colonne de la matrice,

IA est un tableau dont chaque élément contient le numéro de ligne de l'élément correspondant de A.

JC est un tableau dont chaque élément contient le rang, dans la ligne compressée de la matrice d'origine, de l'élément correspondant de A.

Ces tableaux sont surdimensionnés à la taille de la colonne ayant le plus grand nombre d'éléments non nuls. Pour éviter de traiter les cases vides de ces tableaux, on peut leur adjoindre deux masques:

CTXT_A sera un tableau de booléens mise à vrai pour chaque case correspondante non vide des tableaux A, IA et JC.

CTXT_TA sera un tableau de booléens mise à vrai pour chaque case correspondante non vide des tableaux obtenus en transposant la matrice.

Le second masque est notamment utilisé pour obtenir un calcul efficace du produit matrice-vecteur, en parallélisme de données.

L'accès à l'élément M(l, c) de la matrice M, se fera de la façon suivante:

- Recherche de la valeur l dans la ligne IA(c) (Cette recherche peut être obtenue en complexité logarithmique).
- Si l est trouvé à la position IA(i), l'élément cherché est A(i); si l n'est pas trouvé, l'élément cherché est 0.

Le tableau JC, inutile pour l'accès à un élément donné, va permettre de construire rapidement la transposée de la matrice d'origine *(voir l'algorithme de la figure 2.4)*. C'est l'un des intérêts de ce format, par ailleurs plus gros consommateur de mémoire que les précédents.

```
Pour i de 1 à n
        j = 1
        Tant que j ⩽ C et IA(j,i) ≠ 0 faire
                l = JC(j,i)
                c = IA(j,i)
                A_transposée(l,c) = A(j,i)
                IA_transposée(l,c) = i
                JC_transposée(l,c) = j
                j = j + 1
        fin Tant que
fin Pour
```

FIG. 2.4 - *Algorithme de construction de la transposée d'une matrice creuse compressée au format SGP*

Les formats de compression qui viennent d'être exposés sont ceux qui ont été utiles dans les applications numériques abordées dans cette thèse. De nombreux autres formats existent [47, 9], et leur exposé exhaustif déborderait du cadre de ce travail.

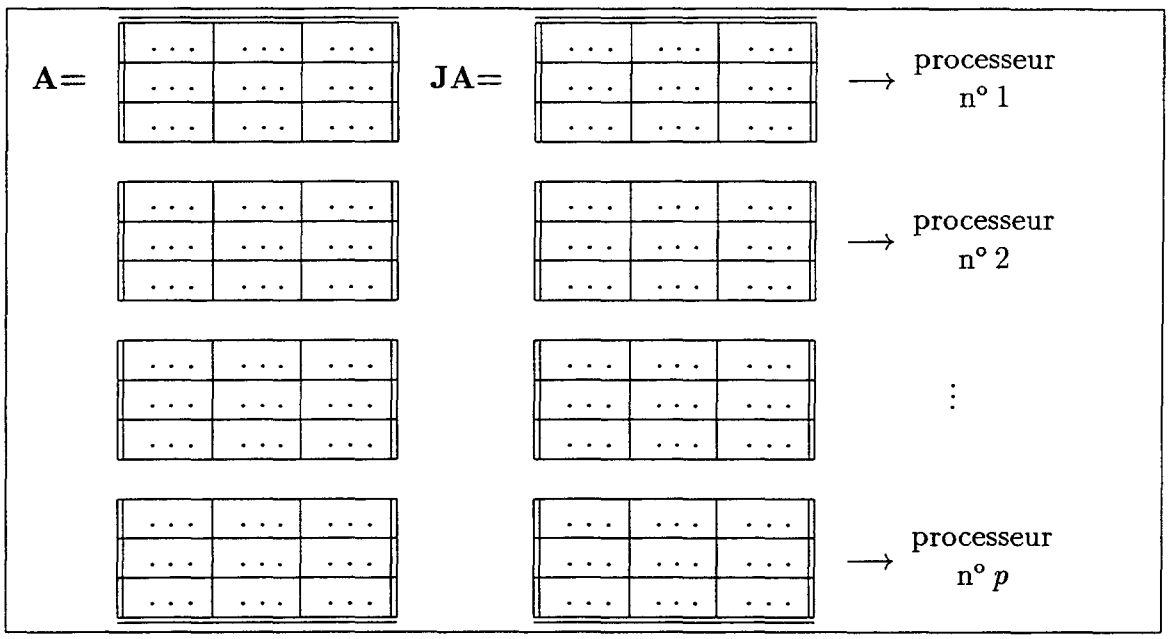


FIG. 2.5 - *Répartition des données d'une matrice creuse au format Ellpack-Itpack sur une machine SPMD*

2.1.3 Parallélisation

Format Ellpack-Itpack

La répartition de la structure de données sur les différents processeurs est particulièrement simple pour une matrice creuse compressée au format Ellpack-Itpack. Il suffit, en effet, de découper les tableaux A et JA en bandes horizontales de même hauteur, chaque bande étant affectée à un processeur *(voir figure 2.5)*.

On peut remarquer que dans cette répartition, chaque ligne est entièrement affectée à un seul processeur, alors que chaque colonne est répartie sur l'ensemble des p processeurs. Il s'ensuit que chaque fois qu'une colonne sera nécessaire au calcul effectué par l'un des processeurs (par exemple dans un produit), les morceaux manquants de la colonne devront être envoyés par les $p-1$ autres processeurs, ce qui induit un nombre de communications important : $p(p-1)$ communications, contenant chacune un morceau de vecteur de longueur $\frac{n}{p}$, sont nécessaires ici pour un produit matrice-vecteur.

Or le produit matrice-vecteur [42] figure dans la majorité des méthodes itératives en algèbre linéaire, et notamment dans l'algorithme GMRES *(voir la section 2.2.1)*.

Formats Ellpack-Itpack et CSR

On peut allier la facilité de découpage de la matrice au format Ellpack-Itpack avec l'économie de place en mémoire procurée par le format CSR en hybridant ces deux formats

de la façon suivante :

- La matrice est générée au format Ellpack-Itpack, et chargée dans ce format par un processus d'initialisation. Ce processus découpe la matrice en bandes destinées aux différents processus de calcul.
- Chaque bande est convertie au format CSR au moment de l'envoi des données aux processus de calcul.

En outre, ce procédé offre l'avantage de réduire la durée de l'initialisation, le volume des données à transmettre étant réduit (le coût de la conversion est négligeable).

Format SGP

Ce format est particulièrement efficace lors du produit matrice-vecteur [9], en parallélisme de données. Soit la matrice M, de taille n, représentée par les tableaux A, IA et JC, de dimensions $c \times n$. Supposons ceux-ci répartis sur une grille 2D de processeurs logiques. Le vecteur V, de dimension n, destiné au produit $A.V$ sera placé initialement selon l'une des lignes de la grille $c \times n$.

Le produit nécessitera donc les opérations suivantes :

- Diffuser chaque élément V_i du vecteur sur tous les processeurs concernant la même colonne de la matrice. Cette opération est parallèle le long du vecteur, et nécessite c communications pour chaque élément V_i. Sa complexité est donc en $\mathcal{O}(c)$
- Faire en parallèle, sur chaque processeur, le produit $M_j^i.V_i$, de complexité $\mathcal{O}(1)$.
- Faire en parallèle la somme des éléments de chaque ligne.

Là se pose un problème, car du fait de la compression par colonnes, les éléments d'une même ligne ne sont pas sur une même rangée de processeurs. Pour les réaligner, on va passer à la représentation SGP de la transposée de la matrice temporaire obtenue par les opérations précédentes *(voir l'algorithme de la figure 2.4)*. Cette transposition est parallèle, à condition que les communications engendrées par les affectations, qui sont des communications globales et non pas de voisinage, puissent se faire simultanément.

L'utilisation du masque CTXT_TA permet d'éviter de traiter les parties inutiles des tableaux, ce qui pourrait avoir un coût non négligeable en cas de sérialisation d'une partie des communications globales, ou des calculs lorsque le nombre de processeurs physiques est inférieur à la grille logique de dimension $c \times n$.

En supposant la granularité maximale, la complexité de la transposition est en $\mathcal{O}(1)$, et la somme des colonnes résultantes s'obtient en $\mathcal{O}(c)$ pour les communications et en $\mathcal{O}(\log_2(n))$ pour les additions.

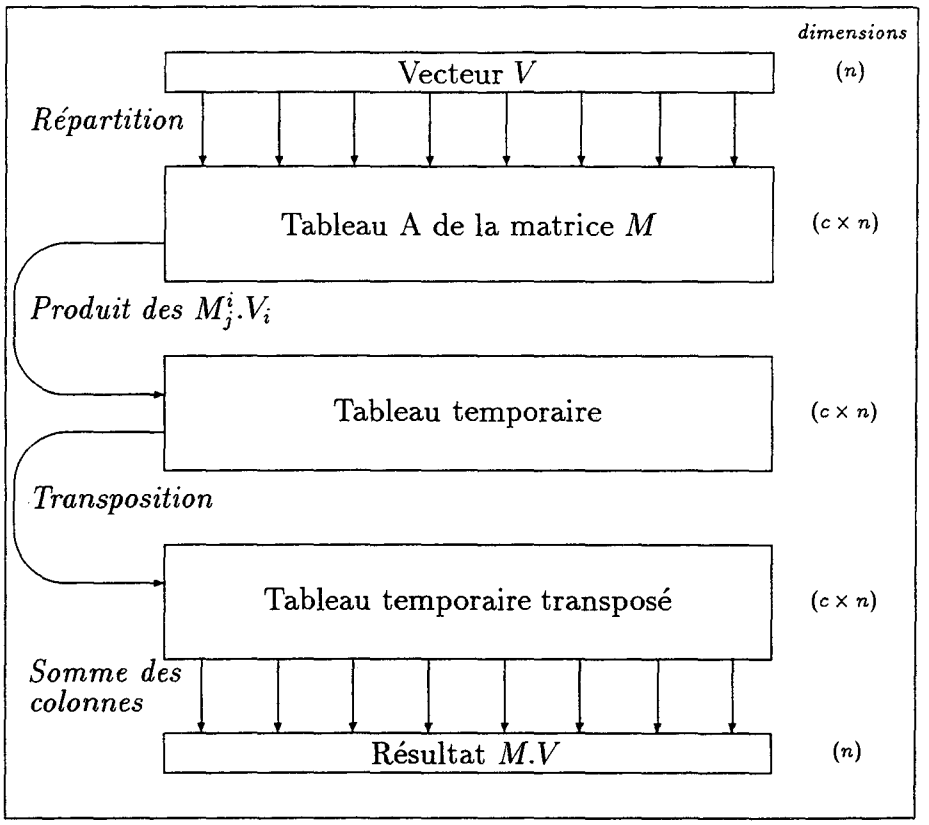


FIG. 2.6 - *Principe du produit matrice-vecteur en format SGP sur une machine à parallélisme de données*

Il est à remarquer que le vecteur résultat a le même placement que le vecteur V original, ce qui est particulièrement intéressant pour les méthodes itératives.

Traitement des blocs denses

La répartition des éléments nuls dans la matrice n'est pas forcément homogène. Le découpage en bandes peut conduire à un déséquilibre dans la charge des processeurs, lors d'un traitement parallèle. Certaines bandes peuvent être très creuses, voire vides, alors que d'autres peuvent provenir de blocs denses de la matrice initiale.

Il est évident que la matrice de l'exemple n° 1 de la *figure 2.7*[1] ne conduira pas à un parallélisme équilibré si elle est découpée en bandes horizontales. L'utilisation du format CSR en affectant à chaque processeur le même nombre d'éléments non nuls à traiter ne résoudra pas complètement le problème, car ce seront cette fois les communications qui seront déséquilibrées lors d'un produit matrice-vecteur (car le nombre de lignes par pro-

1. matrice évoquée lors d'une collaboration avec l'Aérospatiale, et concernant un problème d'électromagnétisme

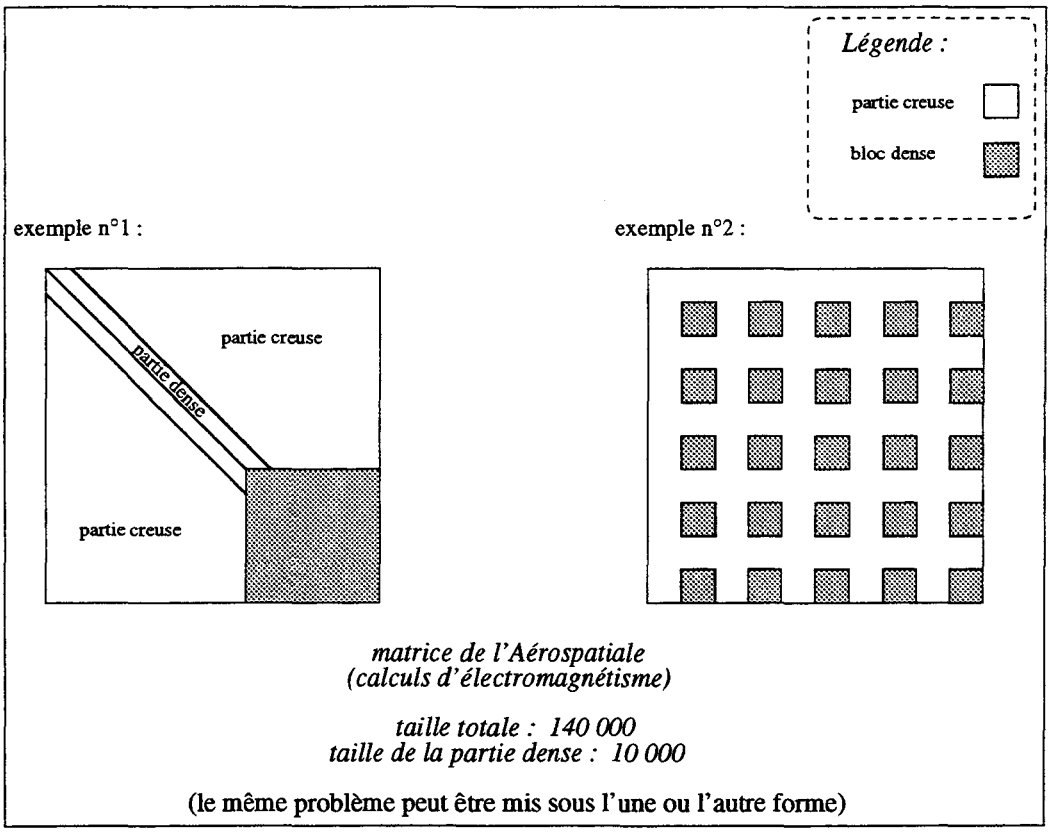


FIG. 2.7 - *Exemples de matrices présentant des blocs denses*

cesseur sera différent). Par contre, la structure de la matrice de l'exemple n° 2 donnera une répartition de charge plus équitable. Une mise en forme préalable peut donc s'avérer indispensable, en présence de blocs denses importants, afin de morceler la partie dense et de rendre la parallélisation plus efficace.

2.2 Méthodes de résolution

Il existe de nombreuses méthodes pour résoudre un système linéaire $A.x = b$.

Les méthodes directes (Gauss, Jordan, LU, ...) ne sont pas idéales pour les systèmes décrits par des matrices creuses de grande taille.

Notamment, elles supportent assez mal les arrondis dûs à la limite de précision liée à la machine. Ces imprécisions peuvent se cumuler au cours des calculs, provoquant des erreurs importantes dans la solution du système. Évidemment, ces problèmes empirent avec l'augmentation de la taille de la matrice.

Un autre problème est la modification, au cours du calcul, de la structure creuse de

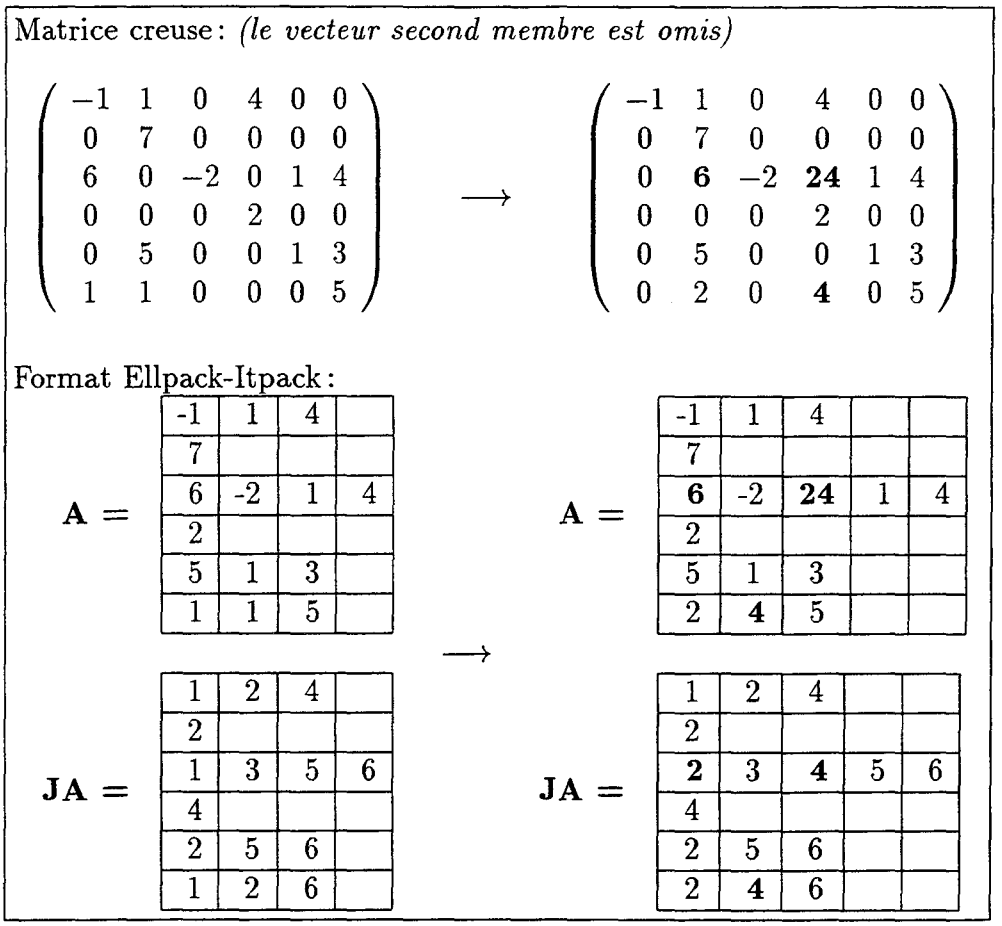


FIG. 2.8 - *Exemple de modification de la structure creuse engendrée par la méthode de Gauss après le traitement de la première colonne*

la matrice. La méthode du pivot de Gauss, par exemple, commence par mettre la matrice sous forme triangulaire supérieure. À chaque étape de ce calcul, on fait apparaître des zéros dans le bas d'une colonne, par un produit matriciel. Celui-ci peut faire apparaître, à la place d'un élément nul de la matrice initiale, un élément calculé non nul *(voir l'exemple de la figure 2.8)*. Dans ce cas, la structure compressée doit être modifiée, pour insérer ce nouvel élément, ce qui peut être coûteux.

Le principe des méthodes itératives est, à partir d'un vecteur initial x_0 plus ou moins aléatoire, de définir une suite de vecteurs $\{x_0, x_1, \ldots, x_k\}$ convergeant vers le vecteur résultat $\bar{x}$ par un calcul répétitif, en calculant à chaque itération i le résidu $b - Ax_i$. Ces méthodes sont beaucoup moins sensibles aux problèmes d'arrondis, et ne transforment pas la matrice initiale A, conservant ainsi la structure creuse et profitant de ses avantages.

La méthode itérative la plus répandue pour résoudre des systèmes linéaires décrits par des matrices creuses non symétriques et de grande taille est GMRES(m), et cette méthode,

ALGORITHME 1 : GMRES

1. Choisir la solution initiale x_0,
   calculer $r_0 = b - Ax_0$, et $v_1 = \frac{r_0}{||r_0||_2}$.
2. Répéter pour $j = 1, \ldots, k, \ldots,$
   $h_{i,j} = (Av_j, v_i), i = 1 \ldots j$
   $\hat{v}_{j+1} = Av_j - \sum_{i=1}^{j} h_{i,j} v_i$
   $h_{j+1,j} = ||\hat{v}_{j+1}||_2$
   $v_{j+1} = \frac{\hat{v}_{j+1}}{h_{j+1,j}}$
3. Établir la solution approchée:
   $x_k = x_0 + V_k y_k$, où y_k est solution de l'*expression (2.4)*.

FIG. 2.9 - *Algorithme n° 1 : La méthode GMRES*

présentée dans la section suivante, a servi de base à la méthode hybride décrite dans cette thèse.

2.2.1 La méthode GMRES(m)

Principe

La méthode GMRES a été introduite par Y. Saad et M. H. Schultz en 1986 [49]. GMRES est l'abbréviation de « Generalized Minimum RESidual ». C'est une méthode itérative qui permet de résoudre efficacement les systèmes linéaires non symétriques de grande taille de type:

$$A.x = b \tag{2.1}$$

où A représente une matrice réelle, b le vecteur second membre, et x le vecteur solution. Cette méthode est particulièrement bien adaptée à la résolution des systèmes décrits par des matrices creuses.

C'est une méthode de projection, basée sur l'algorithme d'Arnoldi [1]. Celui-ci utilise la méthode de Gram-Schmidt pour calculer une base orthonormale $\{v_1, v_2, \ldots, v_k\}$ du sous-espace de Krylov $K_k = \text{span}\{v_1, Av_1, \ldots, A^{k-1}v_1\}$. On obtient ainsi la matrice de Hessenberg $H_k = V_k^T A V_k$ de taille k, et la matrice $\bar{H}_k$, construite à partir de H en lui ajoutant une ligne supplémentaire dont le seul élément non nul est $h_{k+1,k}$, à la position $(k+1, k)$. Les vecteurs v_i et la matrice $\bar{H}_k$ satisfont la relation suivante:

$$AV_k = V_{k+1}\bar{H}_k \tag{2.2}$$

ALGORITHME 2 : GMRES(m)

1. Choisir la solution initiale x_0,
   calculer $r_0 = b - Ax_0$, et $v_1 = \frac{r_0}{||r_0||_2}$.
2. Répéter pour $j = 1, \ldots, m$
   $h_{i,j} = (Av_j, v_i), i = 1 \ldots j$
   $\hat{v}_{j+1} = Av_j - \sum_{i=1}^{j} h_{i,j} v_i$
   $h_{j+1,j} = ||\hat{v}_{j+1}||_2$
   $v_{j+1} = \frac{\hat{v}_{j+1}}{h_{j+1,j}}$
3. Établir la solution approchée :
   $x_m = x_0 + V_m y_m$, où y_m minimise $||\beta e_1 - \bar{H}_m y||_2$, $y \in \mathbb{R}^m$,
4. Redémarrage :
   Calculer $r_m = b - Ax_m$
   Si r_m est satisfaisant alors arrêter,
   sinon réinitialiser $x_0 := x_m$, $\frac{r_m}{||r_m||_2}$ et retourner à l'étape n° 2.

FIG. 2.10 - *Algorithme n° 2 : La méthode GMRES(m)*

Y. Saad et M. H. Schultz ont montré dans [49] que si x_0 est la solution initiale du système, la solution approchée x_k à l'étape k

$$x_k = x_0 + V_k y_k \tag{2.3}$$

peut être trouvée en calculant y_k qui est la solution du problème des moindres carrés

$$\text{minimize}_{y \in \mathbb{R}^k} ||\beta e_1 - \bar{H}_k y||_2 \tag{2.4}$$

où β est la norme résiduelle, c'est à dire $\beta = b - Ax_0$, et e_1 est le premier vecteur canonique de $\mathbb{R}^{k+1}$. L'algorithme GMRES [49] se présente alors conformément à la *figure 2.9 (algorithme n° 1)*.

La valeur de k permettant d'obtenir la solution avec une précision suffisante peut être grande, ce qui implique de pouvoir conserver en mémoire un nombre de vecteurs proportionnel à k et d'exécuter un nombre de multiplications en $\mathcal{O}(k^2)$. Pour remédier à ces inconvénients, Y. Saad et M. H. Schultz ont décrit l'algorithme GMRES(m) [49], qui effectue un redémarrage toutes les m étapes, m étant un paramètre constant défini par l'utilisateur *(voir la figure 2.10 : algorithme n° 2)*.

L'utilisation des transformations de Householder pour résoudre le problème des moindres carrés permet d'obtenir une stabilité satisfaisante [53].

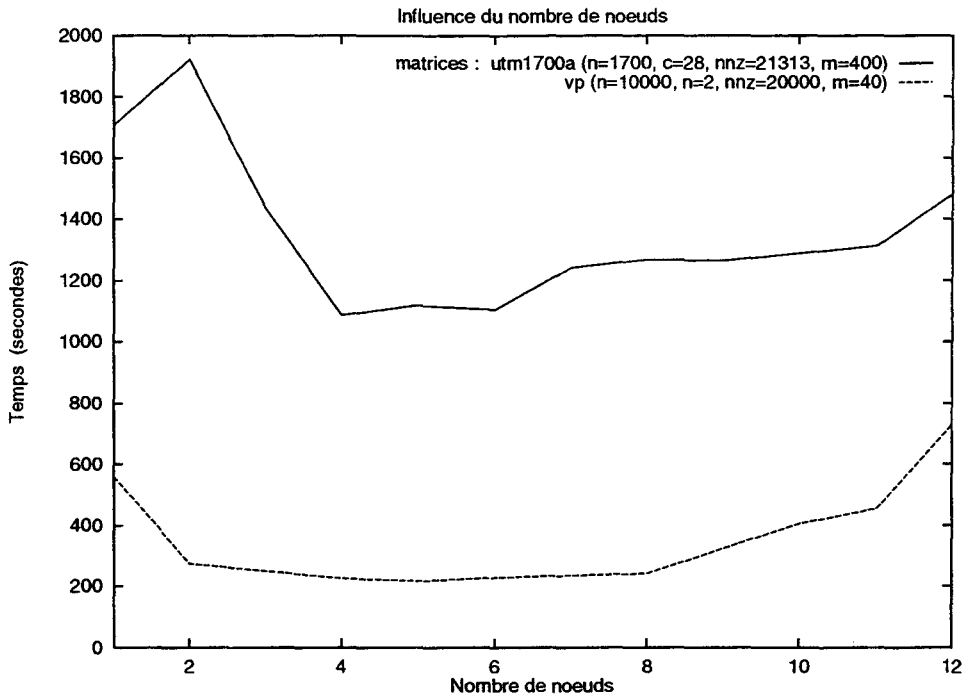


FIG. 2.11 - *Performances de la méthode GMRES(m) parallèle en fonction du nombre de nœuds*

Parallélisation

La méthode GMRES(m) a déjà été portée sur de nombreuses machines parallèles [12]. Mais le gain apporté par sa parallélisation est limitée par les problèmes suivants :

- Alors que certaines parties de l'algorithme, et notamment la projection d'Arnoldi, se parallélisent aisément [40], cette parallélisation est plus difficile pour la factorisation QR concernant la résolution du problème des moindres carrés [34, 39]. Sur une machine SIMD, il est plus intéressant de déporter carrément cette partie du calcul sur une machine séquentielle rapide, d'autant plus que la taille m du sous-espace de Krylov impliqué est toujours très modeste. En parallélisme MIMD, mieux vaut faire ce calcul de façon redondante, afin d'éviter des communications.

- L'augmentation du nombre de nœuds de calcul diminue la charge de chacun d'eux, mais augmente considérablement le nombre de communications [8]. Or GMRES(m) est un algorithme extrêmement communicant : dans une répartition de la matrice par bandes sur une machine MIMD , par exemple *(voir section 2.1.3 et figure 2.2)*, pour chaque produit matrice-vecteur, chaque nœud va devoir recevoir le morceau de vecteur détenu par chacun des autres nœuds. Pour p processeurs, la complexité des communications est en $\mathcal{O}(p)$, alors que celle des calculs est en $\mathcal{O}(\frac{1}{p})$. On comprend dans ces conditions que le degré de parallélisme ne puisse pas dépasser un certain seuil, au delà duquel le coût des communications devient prohibitif *(voir la figure 2.11, les*

matrices utilisées sont décrites dans la section 4.4). À plus forte raison, l'utilisation de plusieurs machines, réalisant du parallélisme de données hétérogène, est illusoire avec cette méthode numérique. Il existe donc, pour une matrice donnée de taille donnée, un degré de parallélisme optimum, et l'amélioration des performances doit être recherchée par d'autres voix que l'ajout pur et simple de nœuds de calcul.

2.2.2 La méthode hybride GMRES/LS-Arnoldi

Principe

Certaines matrices nécessitent un grand nombre d'itérations, par la méthode GMRES, avant d'atteindre la convergence. C'est sur ce nombre que la méthode hybride va jouer, en cherchant à améliorer le vecteur initial x_0 du redémarrage par la connaissance des valeurs propres de la matrice; du moins de certaines d'entre elles car leur calcul exhaustif, si on savait le faire, durerait souvent plus longtemps que la résolution du système linéaire, objet de notre problème.

Nous allons donc employer les nœuds de notre machine parallèle inutilisés par GMRES, ou bien carrément une deuxième machine parallèle reliée à notre réseau, pour calculer un certain nombre de valeurs propres de la matrice par la méthode d'Arnoldi.

Ces valeurs vont être exploitées par la méthode « Least Squares », en tenant compte du vecteur résultat approché déjà obtenu par les itérations précédentes de GMRES, pour calculer un nouveau vecteur initial servant au redémarrage de GMRES.

Le développement de cette méthode hybride et sa mise en œuvre sur machines sont un travail d'équipe interdisciplinaire. En particulier, les aspects numériques du problème ont été étudiés par Azeddine Essai, du laboratoire ANO[2], dans le cadre de sa thèse [13], alors que l'objet de la présente thèse est d'en traiter les aspects informatiques, notamment du point de vue parallélisme et hétérogénéité.

Méthode d'Arnoldi

La méthode d'Arnoldi [1, 45, 48] est une méthode de projection sur un sous-espace de Krylov, permettant d'obtenir de manière itérative une bonne approximation d'un certain nombre de valeurs propres.

L'algorithme de base, dit « projection d'Arnoldi », utilisé dans cette méthode, a d'ailleurs été repris comme un composant de la méthode GMRES *(voir les algorithmes 1 et 2, figures 2.9 et 2.10).* Cette projection de la matrice A de taille n sur l'espace de Krylov $K_m(A, v)$ permet d'obtenir une matrice de Hessenberg supérieure H_m *(voir section 2.2.1),*

2. laboratoire d'Analyse Numérique et d'Optimisation, USTL, Cité Scientifique 59650 Villeneuve d'Ascq, France

ALGORITHME 3 : MÉTHODE D'ARNOLDI

1. Choisir un vecteur initial v de norme 1, la dimension m du sous-espace de Krylov, le nombre désiré d de valeurs propres de plus grand module, avec la précision ε_a
2. Effectuer la projection d'Arnoldi :
   Répéter pour $j = 1, \ldots, m$
   $h_{i,j} = (Av_j, v_i), i = 1 \ldots j$
   $w_j = Av_j - \sum_{i=1}^{j} h_{i,j} v_i$
   $h_{j+1,j} = ||w_j||_2$ si $h_{j+1,j} = 0$ alors arrêter
   $v_{j+1} = \frac{w_j}{h_{j+1,j}}$
3. Calculer les valeurs propres $(\lambda_i, 1 \leqslant i \leqslant d)$ et les vecteurs propres associés $(y_i, 1 \leqslant i \leqslant d)$ de H_m
4. Calculer les vecteurs de Ritz $u_i = V_m y_i$ pour $i = 1, \ldots, d$
5. Calculer les normes résiduelles $\rho_i = ||\lambda_i u_i - Au_i||_2$ $(1 \leqslant i \leqslant d)$
6. Redémarrage :
   si $\max_{i=1}^{d} |\rho_i| < \varepsilon_a$ alors arrêter,
   sinon établir $v = \sum_{i=1}^{d} Re(u_i)$ et aller à l'étape n° 2.

FIG. 2.12 - *Algorithme n° 3 : La méthode d'Arnoldi*

de taille m, dont les valeurs propres sont des approximations des valeurs propres correspondantes de A, et sont appelées les valeurs de Ritz de A.

La qualité de cette approximation augmente avec m [48], mais l'augmentation de la taille de H_m accroît à la fois le coût des calculs (en $\mathcal{O}(m^2)$) et la taille mémoire nécessaire (m vecteurs de taille n, en plus de la matrice H_m). Pour remédier à ces inconvénients, on va là aussi utiliser une stratégie de redémarrage, afin de conserver une valeur de m suffisamment faible. Le vecteur v utilisé comme vecteur initial pour ce redémarrage sera une combinaison linéaire de la partie réelle des vecteurs propres associés à un certain nombre de valeurs propres de plus grands modules. L'algorithme avec redémarrage de la méthode d'Arnoldi est décrit sur la *figure 2.12 (algorithme n° 3)*.

Méthode « Least Squares »

La méthode « Least Squares » [46] est une méthode itérative dans laquelle l'itéré $\tilde{x}$ est défini par

$$\tilde{x} = x_0 + P_k(A) r_0 \tag{2.5}$$

où x_0 représente le vecteur initial, r_0 son résidu ($r_0 = b - Ax_0$), et P_k un polynôme de degré $k-1$.

Ce polynôme sera calculé à partir des coordonnées des sommets du polygone convexe englobant les valeurs propres de A représentées dans le plan complexe, ainsi qu'à partir des paramètres de l'ellipse circonscrite à ce convexe *(voir figure 2.13)*. Cette ellipse possède en effet la propriété d'être symétrique par rapport à l'axe des réels, à condition que la matrice A elle-même soit réelle. Le cas où A est complexe, engendrant une difficulté bien plus importante de la méthode « Least Squares », a été écarté et la méthode hybride GMRES/LS-Arnoldi étudiée ne concerne donc que les matrices réelles.

Il est possible de résoudre le système linéaire uniquement par cette méthode, mais la convergence ne sera atteinte qu'avec la connaissance d'un nombre important de valeurs propres, nécessitant un calcul lourd.

Dans l'hybridation proposée, la méthode « Least Squares » ne sera appliquée que ponctuellement, un nombre déterminé l d'itérations de ce type n'ayant lieu que quand suffisamment de valeurs propres sont disponibles, celles-ci étant calculées à part à l'aide de l'*algorithme 3 (figure 2.12)*.

L'algorithme hybride sera donc celui de la *figure 2.14* (*algorithme n° 4* dans lequel P_k est calculé séparément dès que les valeurs propres ont été obtenues en nombre suffisant).

Si le convexe calculé ne contient que les valeurs propres $\lambda_1, \ldots, \lambda_s$, alors le dernier

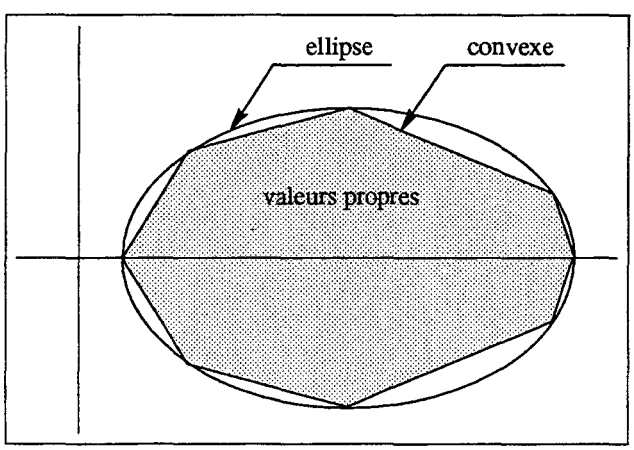


FIG. 2.13 - *Représentation dans le plan complexe des valeurs propres d'une matrice réelle, ainsi que du convexe et de l'ellipse les englobant*

ALGORITHME 4 : GMRES(m)/LS(k,l)

1. Choisir le vecteur initial x_0, la dimension m du sous-espace de Krylov pour GMRES, k le degré du polynôme Least Squares, l le nombre d'itérations Least Squares successives.
2. Calculer x_m, le $m^{\text{ième}}$ itéré de GMRES démarré avec x_0.
   Faire $x_0 = x_m$ et $r_0 = b - Ax_0$
   Si $||r_0||_2 < \varepsilon_g$ alors arrêter.
3. Si un nouveau polynôme P_k est disponible, alors
   faire l'étape n° 4 (Least Squares)
   sinon
   aller à l'étape n° 2 (GMRES)
   fin si
4. Pour $i = 1, \ldots, l$
   calculer $\tilde{x} = x_0 + P_k(A)r_0$
   faire $x_0 = \tilde{x}$ et $r_0 = b - Ax_0$
   fin pour
   Si $||r_0||_2 < \varepsilon_g$ alors
   arrêter
   sinon
   aller à l'étape n° 2 (GMRES)
   fin si

FIG. 2.14 - *Algorithme n° 4 : La méthode hybride GMRES(m)/LS(k,l)*

résidu est conforme à la relation :

$$\tilde{r} = (R_k(A))^l r_0, \tag{2.6}$$

$$= \sum_{i=1}^{s} \rho_i (R_k(\lambda_i))^l u_i + \sum_{i=s+1}^{n} \rho_i (R_k(\lambda_i))^l u_i. \tag{2.7}$$

La première partie du résidu est très réduite car la méthode « Least Squares » trouve R_k qui minimise $|R_k(\lambda)|$ (λ faisant partie du convexe obtenu). Mais pour la seconde partie, le résidu devra beaucoup aux vecteurs propres associés aux valeurs propres situées en dehors du convexe. Ainsi, quand l augmente, la première partie tend vers zéro alors que la seconde partie devient très grande, ce qui explique le fait que le résidu subisse un accroissement si important.

En redémarrant GMRES(m) avec un itéré dont le résidu soit la combinaison d'un petit nombre des vecteurs propres, la convergence sera plus rapide même si le résidu devient provisoirement énorme. C'est pourquoi il est préférable de prendre le résidu calculé après

l'application de la méthode « Least Squares » comme vecteur initial pour les redémarrages de la méthode d'Arnoldi afin de trouver de nouvelles valeurs propres en dehors du convexe calculé précédemment [13]. Cette technique est utilisée dans la méthode « Least Squares-Arnoldi » [48].

Principaux paramètres de la méthode hybride

Les performances obtenues à l'aide de ces méthodes numériques, et notamment de la méthode hybride, sont subordonnées aux paramètres suivants :

n est la taille de la matrice A,

m est la taille du sous-espace de Krylov. Ce paramètre se retrouve aussi bien dans la méthode GMRES qui résout le système linéaire, que dans la méthode d'Arnoldi qui calcule les valeurs propres.. Quand m décroît, la durée de chaque itération diminue, mais le nombre d'itérations nécessaires à la convergence augmente. On peut mettre en évidence une valeur optimale de ce paramètre, qui n'est pas forcément la même dans les deux algorithmes. De plus, il est possible de se servir de ces deux valeurs de m pour régler la vitesse relative des deux calculs, afin d'ajuster le nombre d'envois de valeurs propres pour une efficacité optimale *(voir sections 4.3 et 5.3)*. Il faudra donc distinguer m(**GMRES**) de m(**Arnoldi**).

k est le degré du polynôme « Least Squares ».

l est la puissance du calcul « Least Squares ». Ce paramètre représente le nombre d'itérations « Least Squares » exécutées à chaque prise en compte d'une nouvelle série de valeurs propres supplémentaires.

L'importance des paramètres k et l est étudiée dans la *section 4.4*.

Chapitre 3

Outils pour la programmation parallèle hétérogène

3.1 Parallélisme hétérogène

Lorsqu'on dispose d'un réseau permettant d'accéder à des machines parallèles, il est intéressant de profiter des différentes architectures disponibles afin d'en tirer le meilleur parti pour accélérer les applications les plus lourdes par une parallélisation appropriée. Il est ainsi possible d'ajouter à la puissance de calcul offerte par une machine parallèle, celle offerte par d'autres architectures, séquentielles ou parallèles, reliées par le réseau, y compris en couplant ainsi des machines de générations différentes dont les plus anciennes, souvent délaissées par la plupart des usagers, peuvent encore être d'un appoint non négligeable, à condition de répartir les tâches entre ces machines de manière appropriée.

3.1.1 Répartition des tâches

Pour une application donnée, la première étape consiste en une décomposition de l'application en différentes tâches. Chacune de ces tâches est séquentielle, mais certaines d'entre elles peuvent être exécutées en parallèle. La construction d'un graphe de précédence permettra ensuite de savoir quelles tâches peuvent être exécutées simultanément [38, 7].

Lorsqu'un groupe de tâches est parallélisable, il faudra ensuite savoir quel type de parallélisme adopter, et si plusieurs architectures sont utilisables, quelle sera la répartition la plus efficace en fonction des performances des machines, de l'algorithme utilisé et des communications engendrées.

Pour utiliser au mieux les ressources disponibles, il est souhaitable de mettre d'abord en évidence le « parallélisme de données ». Les tâches opérant sur des structures de données parallèles seront programmées en mettant en œuvre un modèle de programmation approprié à ce type de parallélisme (aboutissant à un code SIMD ou SPMD), et seront affectées

à des machines parallèles. Indépendamment de ce parallélisme de données, on peut souvent mettre en évidence un « parallélisme de tâches », dont chaque composant peut s'exécuter sur une station d'une machines MIMD, voire sur une machine séquentielle, selon le volume des communications s'y rapportant.

Les grosses applications, contenant à la fois du parallélisme de tâches et du parallélisme de données, pourront être réparties sur plusieurs machines parallèles (ou éventuellement séquentielles pour certaines tâches peu communicantes). Il faudra cependant veiller à la cohérence des tâches travaillant sur la même structure de données parallèle, celle-ci devant être répartie sur une seule machine pour éviter des surcoûts de communications. De même, toute paire de tâches échangeant des données volumineuses ou fréquentes aura intérêt à partager la même machine parallèle, son réseau de communication interne étant en général optimisé.

3.1.2 Granularité

La granularité de ces tâches, pour une parallélisation des calculs donnée, va être liée principalement au coût des communications, fonction du placement des données et de l'architecture utilisée. Dans une machine établie en grille ou en réseau de processeurs, les communications de voisinage sont généralement peu coûteuses et il est possible d'obtenir du parallélisme à grain fin ou moyen à condition d'utiliser un placement des données adapté. Par contre, dans les machines construites autour de stations reliées par un réseau, ce qui est une architecture de plus en plus fréquente, toutes les communications sont exécutées par envoi de messages. Même si le réseau de communications interne à la machine est souvent optimisé, elles sont de ce fait beaucoup plus coûteuses, en particulier en comparaison de la puissance des processeurs qui est généralement grande sur ces machines. Le parallélisme sera donc dans ce cas à grain moyen ou gros.

Les communications entre les tâches affectées à des machines différentes devront être minimisées, car elles sont très pénalisantes si les machines sont reliées par un réseau classique (style Ethernet). Le parallélisme entre ces tâches devra donc être à très gros grain.

Lorsque c'est possible, il est donc intéressant dans ce cas d'avoir une gestion asynchrone des processus de communication et de ceux de calcul. La prise en compte des messages arrivés se faisant à certains points particuliers des tâches de calcul. Dans le cas de la méthode hybride présentée dans la *section 2.2.2*, il est notamment possible de recouvrir certaines communications.

3.2 Outils existants

La nécessité d'avoir des outils permettant la réalisation d'applications tirant parti d'architectures hétérogènes est apparue depuis longtemps. Après les outils de base se contentant

d'offrir des primitives de communication et de lancement de tâches sont apparus des systèmes plus élaborés permettant de construire plus efficacement des applications parallèles hétérogènes cohérentes. Mais ces outils restent souvent limités au petit nombre d'architectures pour lesquels ils ont été développés. Voici une description non exhaustive de certains de ces outils.

3.2.1 PVM

PVM (« Parallel Virtual Machine ») est une couche logicielle disponible sous forme de bibliothèque de primitives C ou Fortran, et permettant la communication entre processus, dans un environnement hétérogène [23]. On dispose ainsi :

- de primitives de contrôle dynamique des ressources : des machines (« hosts ») peuvent être ajoutées ou retranchées à tout moment à la « machine virtuelle » définie sous PVM,
- de primitives de gestion de processus, permettant de démarrer ou d'arrêter des tâches,.
- de primitives d'envoi et de réception de message, bloquantes et non bloquantes,
- de primitives de préparation de message et de conversion des données permettant de faire communiquer des machines utilisant des formats de données différents.

PVM dispose aussi d'une « console », c'est à dire d'une interface utilisateur permettant de contrôler l'activité des tâches et leur localisation sur le réseau hétérogène, ainsi que les ressources engagées.

PVM est une bibliothèque extrêmement répandue, et disponible sur un très grand nombre de machines d'architectures différentes. Son principe repose sur l'envoi de messages d'une tâche à l'autre, chaque tâche correspondant à un processus ou à un thread s'exécutant sur une machine donnée. Il s'agit donc d'un outil particulièrement bien adapté au parallélisme de tâches et à des réseaux de machines hétérogènes. Il est notamment possible, grâce à PVM, de faire du parallélisme à peu de frais en tirant parti d'un réseau de machines séquentielles comme s'il s'agissait d'une machine parallèle. Les performances dans ce cas sont cependant limitées par le débit de ce réseau, souvent modeste.

Mais les primitives disponibles sous PVM sont de bas niveau et leur utilisation exige du programmeur une analyse très fine des mécanismes de communication de l'application en question, ainsi que des problèmes de synchronisation liés au parallélisme. Par exemple, Les structures de données distribuées ne peuvent être manipulées que comme une collection de structures de données locales, dont les communications doivent être décrites explicitement.

La bibliothèque MPI, qui concurrence PVM depuis quelques années, a cependant des caractéristiques et des objectifs différents : très optimisée pour être efficace dans les envois

et réceptions de messages, munie d'abstractions de plus haut niveau que PVM, elle est parfaitement adapté au parallélisme de tâches sur un réseau homogène. MPI n'a cependant pas l'universalité de PVM, ni notamment les primitives permettant de faire coopérer des machines utilisant des formats de données différents, et de ce fait n'est pas adapté au parallélisme hétérogène [24].

Beaucoup d'environnements de plus haut niveau font appel aux bibliothèques de PVM ou MPI pour gérer le lancement et l'arrêt des processus, ainsi que les communications. Certains mêmes optimisent les communications en faisant appel, soit à PVM, soit à MPI, soit à d'autres bibliothèques, en choisissant la plus efficace dans chaque cas [20].

3.2.2 Meta-Chaos

Cette librairie a pour but de faciliter le parallélisme de données dans un environnement hétérogène, en permettant la coopération entre plusieurs librairies de communication, dédiées à des machines, à des structures de données, voire à des langages différents.

Dans une application hétérogène lourde, il est fréquent et même souhaitable de réutiliser des éléments de code optimisés, écrits dans divers langages et liés à diverses architectures. Chaque composant logiciel fait appel à une bibliothèque de communication qui n'est pas forcément la même pour tous les composants. De plus, les données sont distribuées selon un modèle qui dépend de la façon dont est établie la structure de donnée dans le langage employé (les tableaux, par exemple, sont gérés en Fortran colonne par colonne, et en C ligne par ligne), et des consignes de placement (« mapping ») données par le programmeur. La librairie Meta-Chaos, présentée par G. Edjlali, A. Sussman et J. Saltz [11], permet à ces différents composants d'échanger des données, en coordonnant les librairies de communication qu'ils utilisent.

Il est ainsi possible, par exemple, de transférer une partie d'un tableau distribué à l'aide d'une librairie *libX* vers une structure de données arborescente distribuée par une librairie *libY*. Le principe utilisé pour définir un « mapping » implicite entre les données d'origine, distribuées par une librairie parallèle, et la destination du transfert, distribuée par un autre librairie parallèle, est nommée par les auteurs « linéarisation ».

Linéarisation

Soit une structure de données A, distribuée par la librairie *libX*, et S_A le sous-ensemble de ces données à transférer vers le sous-ensemble S_B d'une structure de données B distribuée par la librairie *libY* (S_B doit naturellement avoir le même nombre d'éléments que S_A). La linéarisation L_{S_A} peut être vue comme une structure de données abstraite qui procure un ordre total sur les éléments de S_A. Les fonctions de linéarisation ℓ_{libX} telle que $L_{S_A} = \ell_{libX}(S_A)$ et ℓ^{-1}_{libX} telle que $S_A = \ell^{-1}_{libX}(L_{S_A})$ sont fournies par les auteurs de la librairie *libX* (leur écriture nécessite évidemment une connaissance approfondie de la façon

Modèle fully-constrained
with freq(A,c_1:forever:c_2) && freq(B,c_3:forever:c_4) $A = B$
Usage: Programmes fortement couplés.

Modèle producer-constrained
with freq(A,c_1:forever:*) && freq(B,c_3:forever:c_4) $A = B$
Usage: Producteur beaucoup plus rapide que le consommateur.

Modèle consommer-constrained
with freq(A,c_1:forever:c_2) && freq(B,c_3:forever:*) $A = B$
Usage: Consommateur beaucoup plus rapide que le producteur.

Modèle free-running
with freq(A,c_1:forever:*) && freq(B,c_3:forever:*) $A = B$
Usage: Programmes faiblements couplés.

FIG. 3.1 - *Modèles de couplages entre applications hétérogènes mettant en œuvre les « mappings » [44]*

dont la librairie distribue les données).

Le transfert des données de S_A vers S_B peut être vue comme une opération en trois étapes:

1. $L_{S_A} = \ell_{libX}(S_A)$

2. $L_{S_B} = L_{S_A}$

3. $S_B = \ell^{-1}_{libY}(L_{S_B})$

Ce concept de linéarisation présente les propriétés suivantes [11]:

- il est indépendant de la structure des données (le transfert est possible entre deux structures très différentes),

- il ne nécessite pas la description explicite de la redistribution des données entre les deux structures: celle-ci se fait implicitement par le canal des linéarisations respectives des deux structures,

- il s'agit d'une abstraction qui ne coûte rien en ressources mémoires.

Mappings

À partir de ce système Meta-Chaos, une librairie a été établie pour permettre le « mapping » de tableaux à l'usage d'applications parallèles hétérogènes [44]. Il s'agit :

- de relier deux tableaux de mêmes tailles, qui doivent être rendus cohérents selon une fréquence spécifiée (qui correspond au nombre d'exécution de points de synchronisation),
- d'établir ces « mappings » à l'exécution, et de les ajouter ou retrancher dynamiquement par une librairie d'exécution,
- de gérer efficacement le système par la bufférisation et le transfert asynchrone des données, ainsi que par un ordonnancement optimisé.

Une application peut exporter un tableau par la primitive **register**, qui le rend visible aux autres applications et leur permet d'y accéder par un identificateur unique (ce tableau pourra plus tard être rendu inaccessible par **unregister**. Deux primitives permettent d'assurer la cohérence du « mapping » :

acquire Toutes les opérations de cohérence impliquant un tableau pour lequel l'appel à « acquire » a été émis doivent être achevées avant le retour de l'appel à « acquire ».

release Pour un tableau exporté en mode « out », « release » indique qu'une nouvelle version du tableau est maintenant en place, et le restera jusqu'au prochain « acquire » concernant ce tableau.
Pour un tableau exporté en mode « in », « release » indique qu'on peut maintenant sans risque modifier les données dans le tableau.

Ces primitives assurent les garanties de cohérence suivantes :

- Garantie de sécurité de transfert :
  Aucune donnée n'est transférée depuis ou vers un tableau entre deux « acquire/release » associés concernant ce tableau. Les données peuvent être transférées depuis ou vers un tableau à n'importe quel moment entre un appel à « register » et le premier « acquire » ou entre un « release » et le prochain « acquire » (ou « unregister »).
- Garantie de version unique :
  Toutes les données transférées vers ou depuis un seul tableau dans une seule action de cohérence appartiennent à la même version.

Un mapping est établi par une déclaration de la forme :

with *spécification_de_cohérence* { *tableau1* = *tableau2* }.

Un compteur initialisé à zéro est associé à chaque tableau exporté, et est incrémenté à chaque « release » sur ce tableau.
Une spécification de cohérence est de la forme :

freq(*tableau*, *initial*:*final*:*pas*)

Une condition de cohérence intervient quand la valeur du compteur associé à *tableau* appartient à la séquence définie par « initial:final:pas ». Les valeurs prédéfinies **current** (pour « initial »), **forever** (pour « final ») et * (pour « pas ») permettent d'obtenir aisément différents modèles de couplages entre les applications utilisant ces « mappings » *(voir la figure 3.1)*.

3.2.3 Nexus

Le système Nexus, présenté par I. Foster, C. Kesselman et S. Tuecke, est situé à un niveau d'abstraction élevé. Il gère le parallélisme de tâches et les communications s'y rapportant, et offre un système d'exécution à l'usage de compilateurs permettant ce type de parallélisme [18, 19]. Sa conception a été dictée aussi bien par les nécessités du parallélisme de tâches que par le désir de supporter des environnements hétérogènes. Ses caractéristiques sont [21] :

- la gestion de multiples threads de contrôle concurrents,
- un modèle d'exécution piloté par les données,
- l'allocation et la destruction dynamiques (de threads, de variables partagées ou d'autres ressources),
- un espace d'adressage global, basé sur des variables à affectation unique, des pointeurs globaux, ou des canaux de communication,
- l'accès asynchrone aux ressources distantes,
- une exécution séquentielle efficace.

La mise en œuvre de ces principes met en avant les notions de :

- **nœuds**, qui représentent les ressources physiques de calcul engagées ou retirées dynamiquement,
- **contextes**, qui sont des objets comprenant un code exécutable et des données sur un nœud; plusieurs threads peuvent s'exécuter dans le même contexte,
- **pointeurs globaux**, qui permettent notamment de gérer plus facilement les structures de données distribuées,
- **appels à des services distants** *(Remote Service Requests)* : ces appels déclenchent l'exécution d'une fonction d'un contexte désigné par un pointeur global; il s'agit d'une exécution asynchrone, sans valeur retournée à l'appelant, et par conséquent non bloquante.

Naturellement, tout ceci nécessite des structures de programmation particulières, qui se traduisent par un enrichissement des langages des compilateurs faisant appel à Nexus. Deux compilateurs, l'un nommé CC++, l'autre Fortran M, ont été développés comme des extensions respectives des langages C++ et Fortran 77, afin de profiter du système d'exécution de Nexus. Ainsi, Fortran M [17, 22] possède par rapport au Fortran 77 standard les concepts supplémentaires suivants :

- la gestion des tâches est assurée par :
  - les structures *PROCESS*, de syntaxe identique à SUBROUTINE. Les processus ainsi définis sont lancés par *PROCESSCALL* (au lieu de CALL pour les sous-routines),
  - les blocs *PROCESSES ... ENDPROCESSES*,
  - les boucles parallélisées *PROCESSDO ... ENDDO*;
- les communications entre tâches se font au moyen de *channels* et de *mergers* (implantés à l'aide des pointeurs globaux de Nexus) :
  Un *channel* est un canal de communication géré en file d'attente (FIFO) avec un unique émetteur et un unique récepteur; un *merger* est l'équivalent avec des émetteurs multiples,
- le placement des processus peut se faire sur des tableaux de processeurs virtuels, déclarés par la primitive *PROCESSORS*. Ceux-ci seront associés aux processeurs physiques au lancement du programme. Les primitives *LOCATION* et *SUBMACHINE* permettent d'en sélectionner respectivement un, ou un sous-ensemble.

Nexus met en place des mécanismes permettant d'éviter tout non-déterminisme indésirable. Les programmes parallèles engendrés sont donc déterministes par défaut [16].

3.2.4 LC1 et LC2

LC1 et LC2 sont des langages de contrôle permettant de superviser le lancement des tâches, la granularité du parallélisme et les synchronisations. Ils ont été développés pour une machine à mémoire partagée à trois niveaux de mémoire[1], ces trois niveaux étant :

- la mémoire locale aux processeurs élémentaires, organisée en segments, qui constitue la mémoire principale (MP),
- la mémoire partagée, organisée en domaines, qui constitue la mémoire secondaire (MS),
- la mémoire de masse, organisée en fichiers, qui constitue la mémoire tertiaire (MT).

1. le démonstrateur Marianne de Thomson-CSF [38]

LC1

Le principe de la programmation sous LC1 [38] est de décomposer l'application en tâches, en faisant la distinction entre tâches de migration (M_tâches) entre les différents niveaux de mémoire, permettant d'exprimer le parallélisme, et tâches de calcul (C_tâches), ces dernières se trouvant au niveau le plus élémentaire, et étant notamment exemptes de communications. Celles-ci sont gérées dans les M_tâches par les primitives ENVOYER et RECEVOIR agissant sur des structures de données dont les domaines ont été explicités par une déclaration appropriée, selon les modes IN (recopie en mémoire primaire avant la C_tâche), IN_LOCAL (même chose quand les données sont déjà en mémoire primaire), OUT (recopie vers la mémoire secondaire après la C_tâche) ou INOUT (association des modes IN et OUT). La synchronisation est obtenue par l'appel à la primitive ATTENDRE se bloquant sur un booléen (évènement) qui sera mis à vrai à la fin de la C_tâche qu'il représente. Ce système a l'avantage de permettre d'exprimer simplement les dépendances entre tâches, chaque dépendance étant associée à un évènement. Le parallélisme est explicité dans deux structures :

- **PAR** ... // ... **FINPAR** permet d'exprimer que plusieurs parties de programme (séparées syntaxiquement par //) peuvent être exécutées en parallèle, les tâches de chaque branche pouvant être indifféremment des C_tâches ou des M_tâches.
- **POURPAR** $i = a, b$ **FAIRE** ... **FINPOURPAR** indique la parallélisation en $(b - a + 1)$ processus d'une boucle, dont le corps n'est constitué que d'appels de procédures. De plus, les différentes occurences de la boucle doivent pouvoir être exécutées dans n'importe quel ordre.

On peut remarquer que ce langage de contrôle, s'il permet effectivement d'exprimer le parallélisme de tâches, ne concerne ici que la gestion d'une machine parallèle et ne permet pas de prendre en compte l'hétérogénéité. Cependant les principes mis en avant pour la synchronisation et le découpage en C_tâches peuvent être retenus pour le cas hétérogène, et les principes mis en œuvre par LC1 pour gérer les différents niveaux de mémoire distribuée de la machine sont applicables aux machines à mémoire distribuée, la mémoire locale de chaque nœud constituant le niveau n° 1 par rapport à ce nœud, et les mémoires distantes (celles des autres nœuds) formant le niveau n° 2. Il serait même possible de gérer l'hétérogénéité en voyant les mémoires locales des autres machines comme un troisième niveau de mémoire.

LC2

Il s'agit d'un langage de programmation à deux niveaux, dédié au calcul parallèle sur une machine MIMD, et basé sur les concepts suivants [51] :

- une application parallèle est découpée en **tâches**, s'exécutant chacune sur un processeur élémentaire. Ces tâches sont écrites dans un langage de programmation algorithmique quelconque,

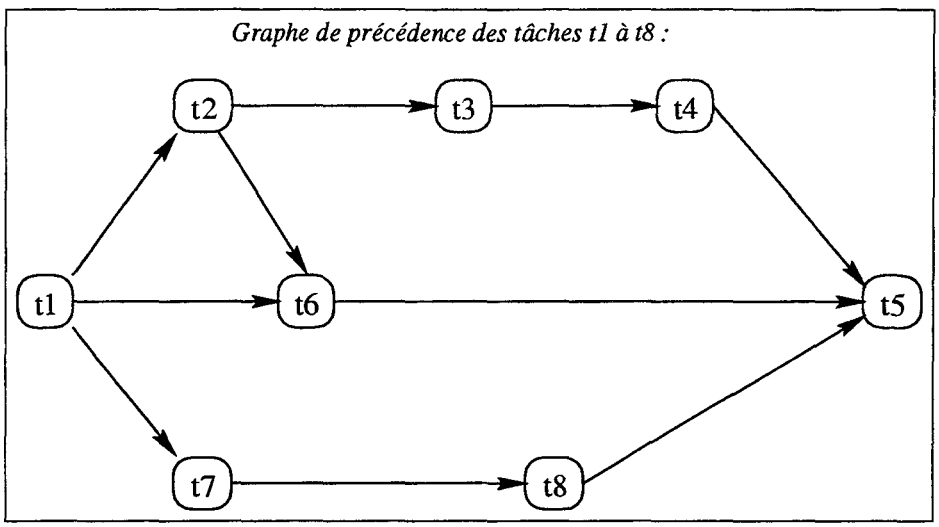
Déclarations des domaines (structures de données en mémoire secondaire) :	
domaine d1 = plein(1:n, 1:p) réel	*Le domaine d1 est un tableau de réels de dimensions n,p*
domaine d2 = plein(1:f) struct a, réel b entier fin	*Le domaine d2 est un tableau de dimension f dont chaque élément contient un réel a et un entier b*
domaine d3 = creux(1:r, 1:s) card z réel	*Le domaine d3 est une matrice creuse de r lignes et de s colonnes possédant z éléments non nuls*
Déclarations des tâches :	
tâche t1(in d1(1:g,h:k), in d3(i,*), out d2.b)	*Avant l'exécution de la tâche t1, les données suivantes sont recopiées de mémoire secondaire (MS) en mémoire principale (MP) :* - *la sous matrice, extraite de d1, dont les numéros de ligne sont entre 1 et g ($1 \leqslant g \leqslant n$) et dont les numéros de colonne sont entre h et k ($1 \leqslant h \leqslant k \leqslant p$),* - *la ligne n° i du domaine creux d2.* *Après l'exécution de la tâche t1, les données suivantes sont recopiées de mémoire principale (MP) en mémoire secondaire (MS) :* - *les f entiers identifiés par b dans le domaine structuré d2.*

FIG. 3.2 - *Exemple de transfert de sous-domaines entre les différents niveaux de mémoire en LC2*

- ces tâches sont coordonnées par un **programme de contrôle**, écrit dans un langage spécialisé (en l'occurence LC2). Ce programme permettra d'exprimer l'organisation des tâches au sein de l'application, ainsi que leur ordre d'exécution. Son rôle sera donc de régir l'ordonnancement des tâches, de tenir compte des dépendances entre elles, et par conséquent de gérer les communications et les synchronisations éventuelles les concernant.

Cette décomposition de l'application parallèle en deux niveaux de programmation présente certains avantages :

- la transcription du graphe est aisée dans un programme de contrôle qui ne reflète que la structure de l'application,



Première solution

```
tâche t1()
PAR
    SYNC e1 PUIS e2
FAIRE
    :e1: tâche t2(); tâche t3(); tâche t4()
//
    :e2: tâche t6()
//
    tâche t7(); tâche t8()
FINPAR
tâche t5()
```

Deuxième solution

```
PAR
    SYNC a3 PUIS a1
    SYNC a3 PUIS a6
    SYNC a1 PUIS a4
    SYNC a2 PUIS a5
    SYNC a7 PUIS a5
FAIRE
    :a1: tâche t2(); tâche t3(); :a2: tâche t4()
//
    :a3: tâche t1(); :a4: tâche t6(); :a5: tâche t5()
//
    :a6: tâche t7(); :a7: tâche t8()
FINPAR
```

FIG. 3.3 - *Exemple de la description d'un graphe de précédence en LC2*

- l'ordonnancement des tâches est facilement modifiable,
- des relations de précédence complexes peuvent être formalisées clairement.

La séquentialité dans l'agencement des tâches est traduite par la notion de flot. Un flot est un ensemble de tâches qui s'exécutent les unes derrière les autres.

Plusieurs flots peuvent être parallèles, et ce parallélisme s'exprimera comme en LC1 par les constructions :

PAR ... // ... FINPAR
POURPAR $i = a, b$ FAIRE ... FINPOURPAR

La syntaxe utilisée par LC2 pour les opérations de transfert des données entre les différents niveaux de mémoire est particulièrement intuitive et d'une grande richesse sémantique. L'information concernant ces transferts se trouve dans les paramètres « in » et « out » des tâches. Une panoplie complète de déclarations d'extraction de domaines permet d'en tranférer des sous-ensembles : sous-matrices, champ d'une structure, diagonale ou transposée d'une matrice. Une simple déclaration des paramètres d'entrée et de sortie d'une tâche générera l'extraction et le tranfert des données utilisées en entrée à l'appel de la tâche, depuis la mémoire secondaire (partagée) vers la mémoire principale du processeur actif pour cette tâche. De même à l'issue de l'exécution de la tâche, les données déclarées en sortie seront tranférées de la mémoire principale vers la mémoire secondaire, où elles seront placées dans les structures de données concernées *(voir la figure 3.2)*.

LC2 se différencie surtout de LC1 au niveau de la synchronisation [38, 51] : notamment, celle-ci ne se fait pas uniquement sur des fins de C_tâches (la notion d'évènement n'existe pas), ce qui permet une granularité plus fine en évitant les surcoûts induits par les changements de contexte lors des appels systématiques aux C_tâches. Cela permet aussi de synchroniser des migrations de données entre les différents niveaux de mémoire, pour éviter des conflits. La synchronisation s'exprime explicitement par des étiquettes, permettant de spécifier un nœud du graphe de dépendance. Il existe même des tableaux d'étiquettes, utiles dans les boucles parallélisées (POURPAR). Ces étiquettes sont disponibles pour exprimer soit un arc de précédence, soit une exclusion mutuelle, par les déclarations suivantes :

- **SYNC a PUIS b** indique une synchronisation de précédence : l'instruction étiquetée par « b » ne pourra s'exécuter qu'après la terminaison de l'instruction étiquetée par « a ». Le connecteur logique ET permet d'exprimer les dépendances multiples,

- **MUTEX a ou b** indique une synchronisation d'exclusion mutuelle : une seule des deux instructions étiquetées par « a » ou « b » ne pourra s'exécuter à un instant donné. Si l'une des deux est en cours d'exécution, l'autre ne pourra commencer que quand celle en cours aura terminé.

Pour tenir notamment compte des cas particuliers des bornes des intervalles, ce qui se produit souvent en parallélisme de données, l'instruction SYNC peut être conditionnelle et s'exprimer par exemple selon le modèle :

$$\text{SYNC } (i = 1, n) \text{ SI } (i > 1) \text{ a}(i) \text{ PUIS b}(i)$$

LC2 est un véritable langage de programmation et permet notamment la décomposition du programme en modules pouvant être compilés séparément, chaque module contenant une ou plusieurs procédures. Ces procédures encapsulent les domaines, les constantes et les variables sur lesquelles elles travaillent (locaux ou passés explicitement en paramètres), et ne peuvent pas accéder aux autres données du programme. Leur appel est synchrone : il n'est terminé que lorsque toutes les tâches, parallèles ou non, appelées par la procédure

sont terminées.

Ces outils permettent d'exprimer des graphes de précédence très complexes. La souplesse du langage permet même souvent de les exprimer de différentes façons *(voir la figure 3.3 dans laquelle la première solution semble plus intuitive que la seconde)*. Mais le programmeur doit être vigilant afin d'éviter des blocages (dont l'exemple trivial est « MUTEX a OU a »), ce qui est particulièrement délicat avec les étiquettes indicées.

3.2.5 High Performance Fortran

Ce language est né sur la demande des industriels (à l'initiative de NEC) de la nécessité de standardisation d'un langage pour le parallélisme de données basé sur le Fortran. Les applications scientifiques étant, pour un grand nombre d'entre elles, rédigées en Fortran, il est intéressant de réutiliser les bibliothèques existantes (séquentielles) pour écrire des programmes parallèles. La possibilité de gérer le parallélisme en Fortran était déjà présente dans la version Fortran 90, et dans des versions constructeurs comme MPFortran (pour la MasPar) ou CMFortran (pour la CM5).

Le High Performance Fortran (HPF) [29, 6] reprend la syntaxe classique du Fortran 77 pour les instructions séquentielles, et s'inspire des constructions parallèles ainsi que des idées de placement des données déjà présentes dans les versions parallèles citées ci-dessus. Naturellement, les structures concernant le parallélisme sont particulièrement riches en HPF. En particulier, le modèle de placement des données est basé sur les notions de « templates », et de « processors », qui représentent des abstractions intéressantes :

- les « templates » sont des tableaux virtuels, formant juste une grille d'alignement, permettant de placer les tableaux de valeurs les uns par rapport aux autres,
- les « processors » sont des processeurs virtuels, permettant au programmeur de définir une granularité maximale, même si elle n'est pas possible sur la machine physique.

Le placement des données se fera donc en trois étapes :

- l'**alignement** des tableaux sur les « templates » (directive ALIGN),
- la **distribution** des « templates » sur les « processors » (directive DISTRIBUTE),
- le **placement** des « processors » sur les processeurs physiques (cette dernière étape ne fait l'objet d'aucune directive).

De très nombreuses possibilités d'alignement et de distribution existent en HPF *(leur description est décrite exhaustivement dans la thèse de F. Coelho* [6]*)*, ce qui donne une grande souplesse pour la recherche d'un code parallèle plus efficace.

Les instructions générant du parallélisme en HPF émanent du parallélisme de données implicitement présent dans les applications traitées habituellement en Fortran, et qui font

un large usage du calcul matriciel. Ainsi les opérations sur des tableaux (ou des parties de tableau (du genre : $A(1:100) = A(1:100) * 5.0$) sont implicitement parallèles. Mais HPF psssède aussi une syntaxe très complète pour exprimer explicitement le parallélisme (en particulier la boucle parallèle FORALL, l'utilisation de masques, les réductions).

Bien que ce langage ne concerne pas directement le parallélisme hétérogène (en particulier, le parallélisme de tâches est marginal dans la définition de HPF), il offre une approche intéressante du placement des données dans un environnement parallèle. Ce modèle pourrait être réinvesti dans un système hétérogène, en étant conscient toutefois que l'optimisation du placement est un problème très complexe, et qu'il l'est encore davantage quand des machines aux architectures et aux performances très différentes sont associées.

3.3 Besoins pour le parallélisme hétérogène

La plupart des outils existants, savent correctement gérer le parallélisme, mais assez peu l'hétérogénéité. Distinguons les besoins liés à ces deux aspects :

3.3.1 Gestion du parallélisme

Tous les langages de contrôle existants possèdent des structures de contrôle autorisant le déroulement parallèle de parties de programme plus ou moins fortement liées les unes aux autres. Il s'agit notamment :

- des blocs parallèles ne comprenant que des appels à des fonctions atomiques, et autorisant le parallélisme à grain fin,
- des boucles parallèles :
  - sans dépendances de données, donnant une parallélisation optimale,
  - ayant au moins une dépendance de données, génératrice de contraintes de synchronisation, et nécessitant des outils permettant d'exprimer ces dépendances.
- des outils de synchronisation (réceptions bloquantes de données, attente de terminaison de tâches).

La prise en compte des dépendance de données est étroitement liée au grain désiré, une application à gros grain distribuée sur des processeurs liés par un réseau aux performances modestes n'ayant de sens qu'avec peu de dépendances.

Un autre type de primitives indispensable concerne le placement des données sur les processeurs. Leur répartition, en fonction des structures de données distribuées et du traitement effectué sur ces données, peut influer considérablement sur les performances [52]. Des langages comme HPF [29, 6] *(voir la section 3.2.5)* donnent des facilités de placement

étendues, mais l'influence de celui-ci sur le comportement de l'application parallèle reste difficile à évaluer.

3.3.2 Gestion de l'hétérogénéité

Lorsqu'on veut faire du parallélisme avec un parc de machines hétérogène, il est particulièrement important de prendre en compte la granularité du parallélisme et de gérer la répartition des tâches à deux niveaux :

- des tâches fortement liées par un parallélisme à grain fin ou moyen, qui doivent impérativement se partager les processeurs de la même machine parallèle.
- des tâches liées aux autres par du parallélisme à gros grain, peu communicantes, et pouvant être déportées sur des machines distinctes, voire couplées par un réseau peu performant.

Les tâches relevant du parallélisme de données pourront être considérées comme des tâches fortement couplées par du parallélisme à grain fin, liées à une machine dédiée à ce genre de parallélisme.

Un problème qui doit être évoqué pour une application hétérogène est celui des entrées-sorties. Il risque même d'être très pénalisant si la structure de données est de taille telle qu'elle ne puisse pas tenir entièrement en mémoire et qu'elle nécessite un traitement « out of core ». Ce problème, très complexe, n'a pas été abordé ici, les applications testées autorisant un seul chargement complet des données au démarrage du programme.

Évidemment, de la structure algorithmique des différentes parties d'une application dépendra son adéquation à une implantation parallèle hétérogène, la fréquence des synchronisations étant déterminante [54].

Dans une application donnée, on pourra considérer plusieurs groupes de tâches, fortement couplées à l'intérieur de chaque groupe (constitué de tâches homogènes implantées sur des machines homogènes), mais liées par un couplage lâche à un autre groupe. Une architecture hétérogène sera alors particulièrement bien adaptée, pour profiter au maximum des ressources disponibles. Un langage de contrôle, autorisant de tenir compte à la fois de la granularité du parallélisme et des architectures disponibles (en définissant des groupes de tâches fortement couplées et en décrivant le type de parallélisme associé, ainsi que les communications entre ces groupes), permettrait de mettre en œuvre plus efficacement les applications potentiellement hétérogènes.

Chapitre 4

Algorithme parallèle hétérogène asynchrone

4.1 Description

Cette première mise en œuvre de la méthode hybride GMRES(m)/LS-Arnoldi décrite dans la *section 2.2.2* utilise au maximum les ressources d'un réseau hétérogène comprenant plusieurs machines parallèles [3, 2, 43]. Elle permet d'avoir un déroulement totalement asynchrone des différents algorithmes, solution qui, si les paramètres sont bien choisis, évite les délais d'attente et offre des performances optimisées. Les particularités de cet asynchronisme seront discutés dans la *section 4.3*.

4.1.1 Organisation générale

L'idée générale est d'utiliser deux machines parallèles, afin que la résolution du système linéaire et le calcul des valeurs propres puissent s'exécuter simultanément de façon indépendante. Les valeurs propres trouvées pourront alors être exploitées de manière asynchrone par l'algorithme itératif de résolution du système pour accélérer sa convergence. Par souci d'efficacité, les parties séquentielles des différents algorithmes seront confiées à des machines séquentielles.

La réalisation de l'implantation de cette version de la méthode hybride [4, 14] utilise deux machines parallèles et deux machines séquentielles :

L'une des machines parallèles va exécuter l'*algorithme 4 (figure 2.14)* mettant en œuvre la méthode GMRES(m)/LS(k, l). Il devra s'agir de la plus puissante des machines parallèles disponibles car la méthode GMRES(m) est celle qui donnera la solution du système linéaire, et dont il convient justement d'accélérer le fonctionnement. De plus, comme cela sera montré dans la *section 4.4*, une vitesse relative trop importante des autres algorithmes détériore les performances.

Une autre machine parallèle exécutera l'*algorithme 3 (figure 2.12)* [1, 9] décrit dans la *section 2.2.2*, qui calculera de manière autonome les valeurs propres de la matrice par la méthode d'Arnoldi. Les différentes parties de cet algorithme sont facilement parallélisables (projection d'Arnoldi, calcul du résidu, redémarrage) à l'exception du calcul des valeurs propres et vecteurs propres par la méthode du QR [34, 9, 39] qui gagne à être déporté sur une machine séquentielle rapide.

Dans la méthode d'Arnoldi, les valeurs propres obtenues au cours d'une itération donnée sont calculées avec une meilleure précision que lors de l'itération précédente. Seules les valeurs propres dont le résidu est inférieur au seuil de précision choisi seront prises en compte. Le choix de ce seuil ε_a est délicat. Si sa valeur est faible, les valeurs propres seront bien calculées et l'efficacité de la méthode hybride sera accrue. Mais une grande précision peut nécessiter un temps de calcul très important et dans ce cas une longue attente des valeurs propres rendra la méthode inopérante. Les valeurs de ε_a donnant une efficacité satisfaisante dépendent de la matrice, et ont varié dans les tests de 10^2 à 10^{-3}.

Il faut aussi choisir le nombre minimum RMIN de valeurs propres, ayant la précision désirée, nécessaire avant l'envoi de ces valeurs pour leur prise en compte dans l'hybridation, et avant le redémarrage de la méthode d'Arnoldi avec un nouveau vecteur initial. Les meilleurs résultats ont été obtenus en choisissant un RMIN petit, et en accumulant les valeurs propres provenant de plusieurs itérations Arnoldi avec différents vecteurs initiaux, plutôt qu'en choisissant des valeurs plus importantes de RMIN.

Parmi les valeurs propres trouvées, seules celles dont la norme résiduelle sera inférieure à la précision ε_a choisie, seront retenues. Quand leur nombre sera suffisant ($\geqslant$RMIN), elles seront envoyées au processus dédié à la partie séquentielle de l'hybridation. Cette partie est exécutée séquentiellement car elle n'utilise qu'un faible volume de données dont la distribution parallèle n'apporterait aucun gain de temps. Il s'agit ici de calculer les paramètres « Least Squares », c'est à dire les paramètres du polygone convexe délimitant la zone contenant les valeurs propres dans le plan complexe *(voir figure 2.13)*, les paramètres de l'ellipse de plus petite aire contenant ce convexe, et les coefficients du polynôme « Least Squares ».

Ce calcul ne va utiliser qu'une partie des valeurs propres, celles trouvées avec une précision suffisante ($< \varepsilon_a$) à l'étape précédente. Le convexe obtenu par ce calcul pourra donc entourer une zone plus petite que le convexe exact. Un résultat satisfaisant ne sera trouvé qu'à partir d'un nombre minimum de valeurs propres transmises.

Ces paramètres seront alors envoyés afin d'exécuter la partie parallèle de l'hybridation. Les processus déroulant l'algorithme GMRES(m) vont recevoir ces données de manière asynchrone, à la fin de l'itération en cours. L'algorithme GMRES(m) sera alors arrêté, et le dernier itéré GMRES(m) ainsi que les paramètres « Least Squares » reçus seront utilisés pour réaliser la partie parallèle de l'hybridation, soit l itérations « Least Squares ». Ensuite, l'algorithme GMRES(m) sera redémarré avec le nouvel itéré x_0 obtenu par la

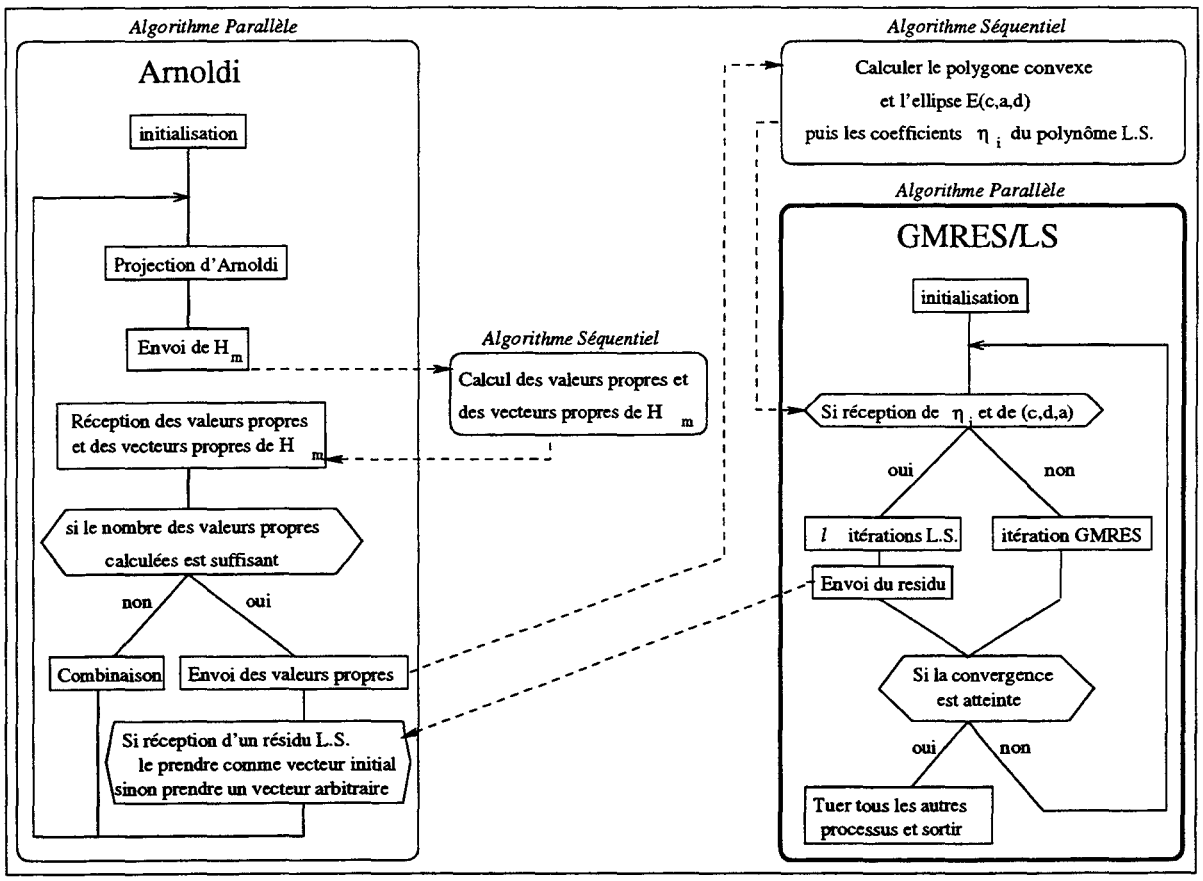


FIG. 4.1 - *Schéma de principe de la méthode hybride GMRES/Least Squares-Arnoldi parallèle hétérogène asynchrone*

méthode « Least Squares ».

Le résidu de ce vecteur sera aussi envoyé à la machine chargée de l'algorithme d'Arnoldi, afin de l'utiliser comme un meilleur vecteur initial pour le prochain redémarrage de cette méthode.

Principales étapes du calcul

En résumé, les principales étapes de la méthode hybride apparaissent sur la *figure 4.1.* Ce sont :

- le calcul des valeurs propres par la méthode d'Arnoldi parallèle, la partie séquentielle du calcul étant déportée pour optimiser les performances,
- le calcul séquentiel des paramètres « Least Squares »,
- la résolution du système linéaire par la méthode GMRES(m) parallèle hybridée par la méthode « Least Squares » à chaque réception des paramètres s'y rapportant.

- Le renvoi du résidu de l'itéré de la méthode « Least Squares » au processus de la méthode d'Arnoldi pour lui servir de nouveau vecteur initial.

Ces étapes sont communes aux différentes implantations de la méthode hybride.

4.1.2 Accumulation des valeurs propres

Afin d'obtenir une meilleure efficacité, il est nécessaire d'accumuler les valeurs propres trouvées pendant plusieurs itérations. En effet, le nombre de valeurs propres utilisables (c'est à dire dont la norme résiduelle est inférieure à la précision demandée ε_a), obtenues après chaque itération de la méthode d'Arnoldi n'est pas toujours suffisant pour définir correctement le convexe de la méthode « Least Squares » ainsi que les paramètres associés. De plus, pour un vecteur initial donné, les valeurs propres obtenues avec une précision suffisante ne sont pas forcément les mêmes d'une itération sur l'autre.

Il faut ajouter qu'il n'est pas possible d'obtenir toutes les valeurs propres de la matrice par la méthode d'Arnoldi. Afin d'obtenir une bonne répartition des valeurs propres calculées, on utilise plusieurs vecteurs initiaux, en changeant de vecteur lors des redémarrages successifs de la méthode d'Arnoldi. En particulier, quand un nouveau vecteur résidu de l'algorithme GMRES/LS est reçu, c'est lui qui va être exploité pour le prochain redémarrage. De cette façon, à mesure des différentes itérations de la méthode d'Arnoldi, un échantillonnage satisfaisant des valeurs propres sera élaboré, permettant de calculer des paramètres cohérents pour la méthode « Least Squares ».

Afin de profiter simultanément des valeurs obtenues après plusieurs redémarrages de la méthode d'Arnoldi, il est important de mémoriser les valeurs trouvées au cours des itérations. Cette accumulation est nécessaire afin d'obtenir un nombre suffisant de ces valeurs, avec une distribution représentative du convexe les contenant toutes. Il sera alors possible de les utiliser efficacement pour élaborer les paramètres « Least Squares ». Toutes ces valeurs mémorisées seront mises à contribution à chaque mise à jour de ces paramètres. Il pourra arriver que des itérations successives de l'algorithme d'Arnoldi redonnent une valeur propre déjà calculée, qui sera alors dupliquée. Mais ces copies ne modifieront pas les caractéristiques du polygone convexe les contenant *(voir la figure 2.13)*.

4.2 Choix d'implantation

4.2.1 Architectures utilisées

Pour expérimenter cette version parallèle hétérogène asynchrone de la méthode hybride, il était important d'utiliser un réseau de machines classique, tel qu'il peut s'en trouver dans beaucoup de centres de calcul, comprenant des machines parallèles non dédiées. Le réseau

du LIFL utilisé pour cette implantation, donne accès à plusieurs types d'architectures parallèles et séquentielles :

- une architecture SIMD avec une MasPar MP1 de 16k processeurs,
- une architecture MIMD avec une ferme d'Alphas DEC de 16 nœuds.
- plusieurs stations de travail SUN pour exécuter les parties séquentielles de l'algorithme.

Les algorithmes GMRES et Arnoldi sont tous les deux des algorithmes utilisant des structures de données régulières (des matrices), et exécutent de nombreuses opérations concernant un sous-ensemble important des données. Ils réclament de très nombreuses communications.

Ces deux algorithmes génèrent donc du parallélisme de données, et peuvent être implantés aussi bien en SIMD qu'en MIMD. À cause du grand nombre de communications engendrées, qui sont la plupart du temps des communications de voisinage privilégiées sur certaines machines SIMD telles que celle dont nous disposons, l'architecture SIMD semble être la mieux adaptée *(voir la section 6.4.1)*. Cependant la puissance de calcul des processeurs disponibles sur les stations MIMD vient concurrencer sérieusement les avantages de beaucoup de machines massivement parallèles handicapées par leurs processeurs trop simples ou trop lents.

D'un point de vue pragmatique, il s'agit, sur un site donné, d'utiliser au mieux les architectures parallèles disponibles afin d'en tirer l'efficacité maximum. Ainsi on pourrait, selon les machines disponibles, utiliser soit deux parties de la même machine parallèle, soit des machines distinctes pour implanter ces deux méthodes numériques.

Avec un réseau disposant des deux types de machines parallèles *(voir figure 4.2)*, le choix a été celui de l'hétérogénéité, afin de pouvoir exécuter simultanément ces deux calculs. L'une des méthodes sera donc implantée sur la ferme d'Alphas, et l'autre sur la MasPar. La ferme, possédant des processeurs plus rapides, accueillera l'algorithme GMRES qui est ici la méthode de base fournissant les solutions du système, alors que la MasPar accueillera l'algorithme d'Arnoldi.

La MASPAR

La MasPar est une machine SIMD munie de 16384 Processeurs Élémentaires 4 bits, pilotés par un processeur unique, l'ACU (Array Control Unit). Ces PEs sont disposés selon une grille torique qui privilégie les communications de voisinage (chaque PE peut communiquer directement avec ses huit voisins). En dehors de ces cas privilégiés, on dispose du Global Router pour faire communiquer deux PEs quelconques, mais avec une durée de communication plus élevée. Cette disposition est importante pour le placement des données

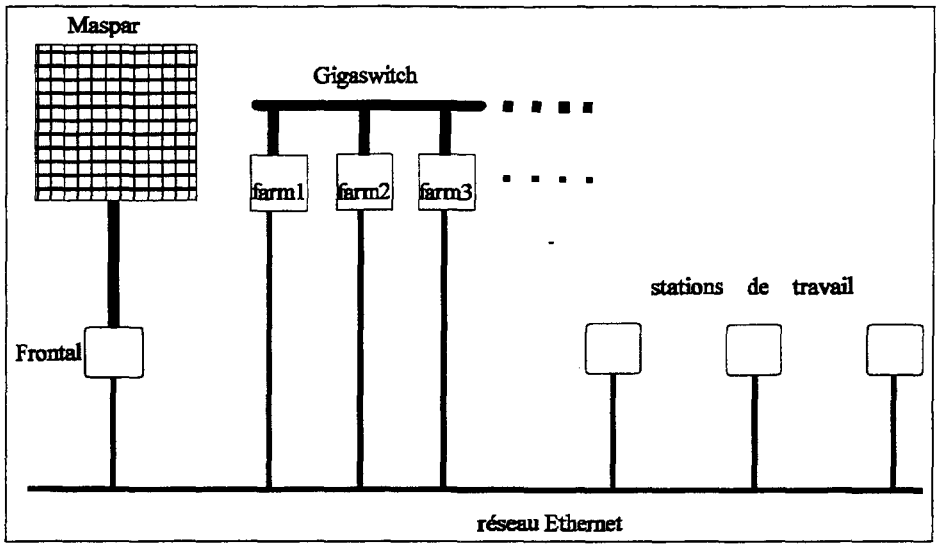


FIG. 4.2 - *Réseau de machines parallèles et de stations de travail disponibles au LIFL et utilisé pour l'implantation hétérogène asynchrone de la méthode hybride*

afin que les communications de voisinage soient privilégiées.

Chaque PE possède une mémoire locale de 64 ko, partagée entre les 4 utilisateurs simultanés possibles, soit 16 ko par utilisateur, ce qui est restrictif lorsque les structures de données sont de grande dimension.

Un ordinateur frontal gère les accès des utilisateurs à la machine ainsi que le transfert des données, et se charge aussi des compilations. Le seul accès possible à ce frontal est le réseau Ethernet, qui constitue un goulot d'étranglement lorsque le volume des données est important.

Un compilateur MPFortran, adapté à cette architecture particulière, est disponible sur cette machine, et a été utilisé pour y implanter la méthode d'Arnoldi. Ce compilateur gère les données parallèles en les répartissant sur les PEs par une simple directive de placement, et les communications entre PEs et avec l'ordinateur frontal sont transparentes au programmeur.

La ferme d'ALPHAs

La ferme d'Alphas est constituée de 16 machines équipées de processeurs Alphas, reliées entre elles par un crossbar à haut débit : le Gigaswitch. Chaque machine est équipée de deux ports de communications : un port Gigaswitch et un port Ethernet.

Aucun ordinateur frontal ne gère l'accès à ces machines et l'utilisation de toute la ferme (ou d'une partie) pour y exécuter un programme parallèle s'obtient en accédant d'abord à

l'une d'entre elles via Ethernet. Chaque machine assure donc à la fois des tâches issues de programmes parallèles et des tâches de compilation, ou de shell d'accueil pour les utilisateurs. A un instant donné, elle peut accueillir un nombre d'utilisateurs très variable, et ses performances peuvent se dégrader énormément si la charge est importante. Dans un programme parallèle ayant des tâches fortement synchronisées, une seule station très chargée peut détériorer l'ensemble des performances, les processus des autres stations attendant l'arrivée au point de synchronisation du processus de la station la plus lente.

Pour faire du parallélisme de données sur cette machine, le programme a été écrit sous forme de processus SPMD, utilisant Fortran 77 et PVM dans un souci de portabilité.

4.2.2 Gestion des communications

Les langages utilisés sur les différentes machines engagées ne savent gérer que les processus et les communications propres à la machine sur laquelle ils sont installés. Pour gérer les échanges entre machine, lancer les processus distants et les synchroniser, il faut faire appel à des bibliothèques de communication (PVM ici car c'était la seule bibliothèque disponible sur toutes les machines concernées). Ces bibliothèques sont employées :

- au sein de l'algorithme MIMD utilisé sur la ferme d'Alphas, pour gérer les échanges de données effectuées par envoi de messages,
- au niveau hétérogène, pour gérer la coordination des processus, et les communications entre les différentes architectures engagées.

4.2.3 Répartition des processus

La méthode hybride, implantée sous sa forme hétérogène asynchrone, va donc être gérée par différents processus s'exécutant simultanément sur différentes machines *(voir figure 4.3)*.

La ferme d'Alphas aura en charge l'*algorithme 4* (GMRES(m)/LS(k,l) : *voir page 33*) sous forme SPMD : Un processus gestionnaire lancera p processus identiques, sur p stations, et distribuera les données de la matrice entre ces processus. Ceux-ci recevront $\frac{n}{p}$ lignes creuses de la matrice et exécuteront la partie de la méthode GMRES(m) concernant leur propre sous-matrice, communicant avec leurs processus frères par le Gigaswitch de la ferme d'Alphas. Les lignes des matrices testées ayant en moyenne le même nombre d'éléments non nuls, la charge se trouve équilibrée (un équilibrage plus fin de la charge n'a pas été traité dans cette première version, mais le problème général de l'adaptativité à la charge des machines sera abordé au *chapitre 7*). La partie séquentielle de cet algorithme (basée sur la factorisation QR) sera faite de manière redondante sur chaque processeur. Une parallélisation de cet algorithme serait possible, mais ne serait pas justifiée ici : les calculs de cette section ont une complexité en $\mathcal{O}(m^3)$, mais la taille m de la matrice H

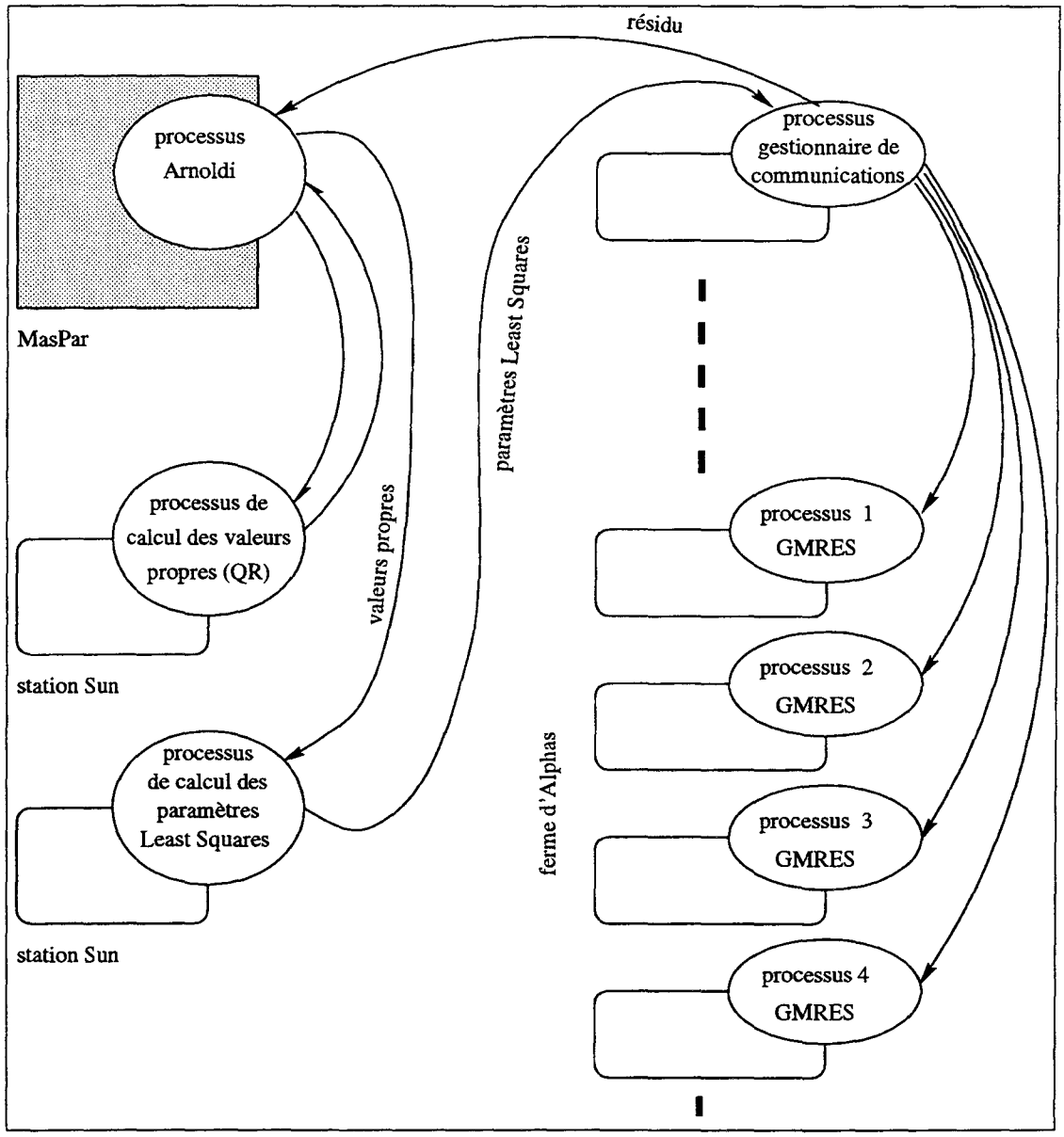


FIG. 4.3 - *Schéma général de l'implantation parallèle hétérogène asynchrone de la méthode hybride GMRES/Least Squares-Arnoldi*

concernée est petite ($m \ll N$) par rapport à la taille N du système à résoudre. Chaque station de la ferme d'Alphas étant aussi performante qu'une machine séquentielle rapide, cette redondance n'est pas pénalisante par rapport à un tranfert sur une station séquentielle, et elle évite même des communications coûteuses.

La MasPar recevra la partie parallèle de l'algorithme d'Arnoldi, comprenant la projection d'Arnoldi, le calcul du résidu, et le redémarrage. La partie séquentielle de cet algorithme (les méthodes QR et de l'itération inverse, et le tri des valeurs propres) est

Paramètres liés aux méthodes numériques :

n	taille de la matrice,
m(GMRES)	taille de l'espace de projection pour la méthode GMRES(m),
m(Arnoldi)	taille de l'espace de projection pour la méthode d'Arnoldi,
k	degré du polynôme « Least Squares »,
l	nombre d'itérations « Least Squares » après chaque prise en compte des paramètres calculés pour cette méthode.

Paramètres liés à la mise en œuvre :

c	hauteur des tableaux SGP (nombre maximum d'éléments non nuls par colonne),
p	nombre de processeurs engagés pour exécuter les processus GMRES/LS,
ε_g	valeur limite de la norme résiduelle pour la convergence de la méthode GMRES(m),
ε_a	valeur limite de la norme résiduelle pour le calcul des valeurs propres par la méthode d'Arnoldi,
ITERMIN	nombre minimum d'itérations GMRES(m) entre deux prises en compte de paramètres « Least Squares » (afin d'éviter une divergence : voir la *section 4.4*),
RMIN	nombre minimum de valeurs propres à trouver par la méthode d'Arnoldi avant de changer de vecteur initial,
NVPMIN	nombre minimum de valeurs propres à accumuler avant de calculer les paramètres « Least Squares ».

Paramètres liés à l'architecture :

NB_LINES	nombre de lignes de la matrice distribuées sur les processeurs n^{os} 1 à $p-1$ pour les itérations GMRES(m)/LS,
NB_LAST	nombre de lignes de la matrice sur le processeur n° p pour les itérations GMRES(m)/LS (NB_LAST $\neq$ NB_LINES si n n'est pas divisible par p).

FIG. 4.4 - *Principaux paramètres de l'implantation parallèle hétérogène asynchrone de la méthode hybride GMRES(m)/LS-Arnoldi*

déportée sur une station SUN, plus rapide que le frontal de la MasPar.

Une autre station SUN calcule les paramètres « Least Squares » (les sommets du polygone convexe, les paramètres de l'ellipse et les coefficients du polynôme « Least Squares »), qui seront ensuite envoyés à la ferme d'Alphas. Ces paramètres transiteront par le réseau Ethernet jusqu'au processus père GMRES/LS, qui servira aussi de gestionnaire des communications. Celui-ci réalisera alors un « multicast », par le Gigaswitch, de ces paramètres vers les processus exécutant la méthode GMRES(m). Les avantages de l'existence de ce processus intermédiaire sont :

- que le processus d'Arnoldi ne connaisse qu'un seul interlocuteur.
- que les données ne transitent qu'une fois par le réseau Ethernet (leur distribution utilise le Gigaswitch de la ferme d'Alphas, beaucoup plus rapide).

- que la réception de ces données par les processus frères SPMD exécutant chacun l'algorithme GMRES/LS sur la sous-matrice qui leur a été distribuée soit synchronisée, afin que la prise en compte intervienne après le même nombre d'itérations GMRES(m) pour chaque processus, et que la structure de donnée distribuée reste cohérente.

A la fin de chaque itération GMRES(m), les processus regarderont si des paramètres « Least Squares » sont arrivés. Dans ce cas, ceux-ci seront pris en compte pour exécuter l itérations de la méthode « Least Squares » avant de faire un redémarrage de GMRES(m). Le résidu du vecteur itéré obtenu à ce moment sera envoyé au processus gestionnaire de la ferme, qui le transmettra à la MasPar pour obtenir un meilleur vecteur initial pour le prochain redémarrage de l'algorithme d'Arnoldi.

Les principaux paramètres d'implantation de cet ensemble de processus sont décrits par la *figure 4.4*. L'influence des plus importants d'entre eux est discutée dans les *sections 4.3.3, 4.4.3 et 4.4.4*. Une approche pragmatique du réglage de ces paramètres est exposée dans la *section 6.3*.

4.2.4 Mise en place de la matrice

Le stockage de matrices de grande taille pose de sérieux problèmes d'occupation des disques. C'est pourquoi la méthode a d'abord été testée avec des matrices générées dynamiquement à chaque lancement du programme. Pour valider le système, des matrices industrielles ont aussi ensuite été importées et stockées *(voir section 4.4)*. Dans les deux cas, la matrice est toujours générée ou lue simultanément sur les deux machines parallèles, afin de limiter les échanges de données par le réseau (tout au moins dans le cas d'une matrice générée). Cette duplication des données, coûteuse en espace mémoire, présente l'avantage de rendre complètement indépendantes les deux parties de l'application, et de permettre un fonctionnement totalement asynchrone de l'ensemble.

Comme il s'agit de matrices creuses, les formats de compression utilisés sont le CSR sur la ferme d'Alphas, pour réduire la taille des messages lors de la distribution des données, et le format SGP sur la MasPar, pour profiter des avantages de ce format lors du produit matrice-vecteur en parallélisme de données *(voir section 2.1.3)*.

4.3 Asynchronisme

4.3.1 Non déterminisme

Les machines qui accueillent les différents processus de la méthode travaillent en temps partagé avec d'autres utilisateurs. Par conséquent la charge de travail de chacune d'entre elles évolue indépendamment de celle des autres, et de façon imprévisible au cours du temps. Pour une exécution donnée, le calcul hybride va coupler les valeurs propres de

l'itération n° a_1 de l'algorithme d'Arnoldi implanté sur la MasPar, avec l'itéré n° g_1 de l'algorithme GMRES(m) implanté sur la ferme d'Alphas. Si on recommence l'exécution du programme avec la même matrice et sans avoir changé aucun paramètre, les valeurs propres issues de la même itération Arnoldi n° a_1 seront couplées cette fois avec l'itéré de GMRES(m) n° g_2 ($\neq g_1$), du fait que les charges de travail et donc les vitesses des différentes machines n'auront pas évolué de la même façon *(voir figure 4.5)*. Il en résulte un non déterminisme du calcul hybride et une non reproductibilité des valeurs de contrôles affichées. Naturellement, ceci est sans incidence sur le résultat numérique final, à savoir les solutions du système linéaire, mais l'accélération de la convergence due à l'utilisation de la présente méthode hybride peut en être affectée.

4.3.2 Précautions nécessitées par l'asynchronisme

La réception des données étant asynchrone, il peut arriver que les valeurs de l'itération n° i arrivent avant celles de l'itération n° $i-1$. Un estampillage des messages est donc nécessaire pour éviter toute confusion (notamment au niveau des échanges entre processus SPMD au cours des itérations « Least Squares », ainsi que pour la réception des paramètres calculés à partir des valeurs propres.

Une barrière de synchronisation doit aussi être mise en place pour la prise en compte des paramètres de la méthode « Least Squares » par les processus SPMD exécutant les itérations GMRES/LS, afin que cette prise en compte, quand elle a lieu, soit validée par tous les processus frères au même numéro d'itération. En effet, malgré la présence de

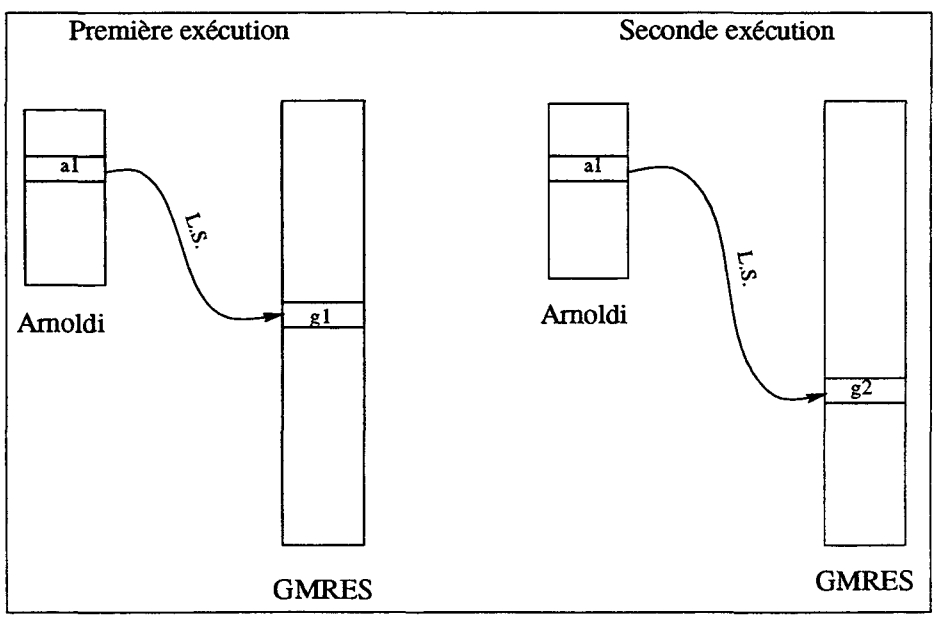


FIG. 4.5 - *Asynchronisme des processus pour la prise en compte des valeurs propres dans la méthode hybride asynchrone hétérogène parallèle*

plusieurs synchronisations par itération GMRES(m) dues aux communications bloquantes (échanges des parties de v, h^j, x détenues par chaque processeur aux étapes n^{os} 2 et 3 de l'*algorithme 2*), la réception des paramètres de la méthode « Least Squares » peut se trouver décalée d'un processus sur l'autre, soit par une durée d'acheminement différente du message, soit par une vitesse d'exécution différente des processus. Il suffit que le message arrive avant l'instruction de réception pour un processus, et après pour un autre, pour créer une distorsion. Les processus ayant reçu les paramètres vont alors se lancer dans une itération « Least Squares » alors que les autres feront une itération GMRES(m). Évidemment, le prochain échange de message se soldera par un blocage (« dead-lock »), les identifiants (« tag ») des messages émis et attendus étant incompatibles. D'où la nécessité de mettre en place un mécanisme pour synchroniser la réception de ces paramètres.

4.3.3 Synchronisation et convergence

L'accélération de la convergence, recherchée par cette méthode car la diminution du nombre d'itérations permet de réduire la durée du calcul d'une manière significative, est subordonnée à la répartition des valeurs propres trouvées par la méthode d'Arnoldi, et aux résultats approchés des itérations GMRES(m) précédentes. Le choix du meilleur moment pour le couplage tel que l'accélération soit optimale, est un problème ouvert. Cependant certains paramètres ont une influence déterminante, ainsi que la décrivent les paragraphes suivants.

Précision du calcul des valeurs propres

La précision désirée ε_a pour l'obtention des valeurs propres influe à la fois sur l'efficacité de l'hybridation « Least Squares » et sur la durée du calcul des valeurs propres. En effet, plus l'exigence dans cette précision est grande, plus il faudra d'itérations à l'algorithme d'Arnoldi pour calculer un nombre suffisant de valeurs propres pour obtenir des paramètres « Least Squares » significatifs, ce qui induira une première prise en compte plus tardive de ces paramètres, puis une fréquence des itérations « Least Squares » plus réduite. Si au contraire ε_a ne correspond qu'à une précision médiocre, on obtiendra rapidement beaucoup de valeurs, mais celles-ci seront moins bien calculées et les paramètres déduits seront moins efficaces.

Par conséquent, l'instant du couplage ainsi que sa fréquence vont dépendre de cette précision. Pour une matrice donnée, il n'est pas évident que la meilleure accélération de convergence soit obtenue avec l'exigence la plus grande quant à la précision des valeurs propres. Les tests ont même permis d'observer que l'efficacité de la méthode hybride était meilleure quand la première itération « Least Squares » intervient assez tôt. Les matrices testées ont révélé de très grandes différences de l'une à l'autre quant à la valeurs de ε_a donnant les meilleures performances (de 10^{-3} à 10^2 !).

Tailles des espaces de projection

Les tailles m(GMRES) et m(Arnoldi) choisies pour les espaces de projection des méthodes GMRES(m) et Arnoldi vont influer sur le couplage, en agissant elles aussi sur les temps de calcul et l'efficacité de la prise en compte des valeurs propres. En effet, le volume des calculs, et donc la durée d'une itération dépend de m pour chacune de ces deux méthodes. La convergence de la méthode hybride, influencée par les instants des couplages correspondant aux prises en compte des valeurs propres par la méthode « Least Squares », va dépendre aussi des valeurs relatives de ces deux tailles.

Si m(Arnoldi) est trop petit, le calcul des valeurs propres sera rapide mais on n'en obtiendra que peu et le convexe calculé alors par la méthode « Least Squares » sera très éloigné du convexe idéal contenant toutes les valeurs propres. D'où des prises en compte des paramètres « Least Squares » rapprochées mais peu efficaces. Au contraire, si m(Arnoldi) est grand, on obtiendra beaucoup de valeurs et le convexe calculé se rapprochera du convexe idéal. Mais le calcul des valeurs propres, qui utilise une matrice de Hessenberg de taille m(Arnoldi), sera alors lent car il devra traiter un volume de données important, et la partie séquentielle du calcul pourrait même risquer de constituer un goulet d'étranglement. Pour m(Arnoldi) élevé, les itérations « Least Squares », synonymes du couplage entre les composants de la méthode hybride, seront donc espacées. Naturellement, si elles le sont trop, elles n'auront plus qu'un effet réduit sur la convergence. En poussant chacun de ces deux cas à l'extrême, l'influence de l'hybridation en sera réduite (soit à cause d'un couplage fréquent mais peu efficace, soit en raison d'une fréquence de couplage insuffisante), et les performances de rapprocheront de celles de la méthode GMRES(m) pure *(voir la figure 4.14)*.

Les valeurs de m(Arnoldi) et de m(GMRES) devront donc être choisies pour obtenir une efficacité optimale. Mais ce choix n'est pas simple, car il dépend de la matrice utilisée dont il n'est pas facile, à priori de prévoir les meilleurs paramètres. Les valeurs de m(GMRES) et m(Arnoldi) utilisées au cours des tests décrits dans la *section 4.4* ont été déterminées empiriquement en choisissant les valeurs donnant les meilleures performances au cours de tests préliminaires.

4.4 Résultats

Les performances relevées sur les tests avec la méthode hybride ont été systématiquement comparées avec celles de la méthode GMRES(m) pure, dans les mêmes conditions de parallélisme. Une répartition sur huit processeurs du calcul GMRES(m) pur ou GMRES(m)/LS a été systématiquement employée, car donnant la meilleure efficacité *(voir la figure 2.11)*.

Aucun préconditionnement n'a été employé, car il n'aurait pas été possible d'utiliser

le même, avec la même efficacité, sur les différentes matrices, et il paraît plus rigoureux d'employer toujours la même méthode de référence. En outre, mettre en œuvre un préconditionnement aurait compliqué considérablement le problème, en ajoutant des paramètres délicats à optimiser, sans compter que ce préconditionnement aurait dû être établi par un calcul supplémentaire, précédent l'ensemble des calculs ici décrits. Il est bien évident que la méthode hybride, complexe à mettre en œuvre, s'adresse avant tout à des cas où la convergence est difficile, ou pour lesquels on ne connaît pas de préconditionnement réellement efficace.

4.4.1 Exemples choisis pour les tests

Plusieurs systèmes linéaires ont été testés pour comparer l'algorithme hybride avec la méthode GMRES(m) pure. Afin de pouvoir vérifier l'exactitude des valeurs numériques des solutions trouvées par l'algorithme, le vecteur second membre b a été choisi tel que la solution du système soit $x = (1, 1, \ldots, 1)^T$ (on a alors $b = \sum_{i=1}^{n} A_i$). Les méthodes démarrent avec $x_0 = (0, 0, \ldots, 0)^T$.

Premier exemple(« vp »): Soit la matrice A de taille $2q \times 2q$ présentée sur la *figure 4.6.* Ses valeurs propres sont $a_j \pm ib_j$, et on prendra les valeurs propres dans deux rectangles R_1 de sommets $-2.5 \pm 2i$, $-1.5 \pm 2i$, et R_2 de sommets $1.2 \pm 4i$, $1.8 \pm 4i$.

$$A = \begin{pmatrix} a_1 & \frac{b_1}{2} & & & & & & \\ 2b_1 & a_1 & & & & & & \\ & & a_2 & \frac{b_2}{2} & & & & \\ & & 2b_2 & a_2 & & & & \\ & & & & \ddots & & & \\ & & & & \ddots & & & \\ & & & & & & a_q & \frac{b_q}{2} \\ & & & & & & 2b_q & a_q \end{pmatrix}$$

FIG. 4.6 - *Exemple de matrice creuse utilisée pour tester la méthode (matrice « vp »)*

Deuxième exemple(« utm1700a »): Des matrices industrielles, issues du serveur Web « MatrixMarket »[1] [5] ont aussi été essayées, notamment la matrice « utm1700a » utilisée en physique nucléaire (taille 1700x1700, 21313 éléments non nuls)[2] Les tests effectués ont montré une convergence difficile de GMRES(m) pur avec cette matrice, lorsque la taille m est modeste *(voir figure 4.9).* D'où l'utilisation de $m = 400$ dans les autres courbes.

1. http://math.nist.gov/MatrixMarket/
2. tous les détails concernant cette matrice peuvent être trouvés sur :
http://math.nist.gov/MatrixMarket/data/SPARSKIT/tokamak/utm1700a.html

GMRES(m) pur: (n=1700, m(GMRES)=400)
- convergence après 10 itérations
- durée 914 secondes

Résultats de la méthode hybride:
(n=1700, m(GMRES)=400, m(Arnoldi)=256, k=15, l=5)
(chaque colonne représente un nouvel essai)

Nombre d'itérations GMRES(m)	6	6	6	7	6
Nombre d'itérations « Least Squares »	2	1	2	3	2
Durée totale (en secondes)	574	544	611	734	684

FIG. 4.7 - *Non déterminisme de l'accélération de la convergence (matrice « utm1700a »)*

Troisième exemple(« 5-points »): Cet exemple provient de la bibliothèque SPARSKIT [47]. La matrice est issue de la discrétisation du problème d'équation aux dérivées partielles suivant:

$$-\Delta u + 10^6 u_x + \frac{\partial}{\partial y}(10^4 e^{-xy} u) = f$$

Elle présente l'avantage d'être paramétrable et de pouvoir être générée facilement à chaque lancement de la méthode de résolution.
Elle donne une stagnation du résidu avec GMRES(m) pour le second membre choisi précédemment pour l'ensemble des matrices testées (*voir la figure 4.11*). C'est pourquoi un autre second membre a aussi été essayé avec cette matrice: $b = (1, 0, \ldots, 0)^T$.

4.4.2 Accélération de la convergence

Les résultats expérimentaux montrent que l'accélération de la convergence est significative, mais dépend beaucoup de la matrice utilisée *(voir les figures 4.8, 4.9, 4.11 et 4.12)*, et d'un choix judicieux des paramètres *(voir notamment la section 4.3.3 à propos de l'influence de* m(GMRES) *et* m(Arnoldi)*)*. En particulier, les paramètres k (degré du polynôme « Least Squares ») et l (nombre d'itérations « Least Squares » à chaque couplage hybride) quantifient l'interaction des méthodes numériques de base au sein de la méthode hybride *(voir la section 4.4.3)*. On observe que le temps de traitement global de la méthode hybride asynchrone hétérogène parallèle est pratiquement toujours meilleur, et souvent de loin, que celui de la méthode GMRES(m) pure parallèle. L'accélération peut même être particulièrement spectaculaire quand la convergence de la méthode GMRES(m) pure est difficile *(voir la figure 4.10)*.

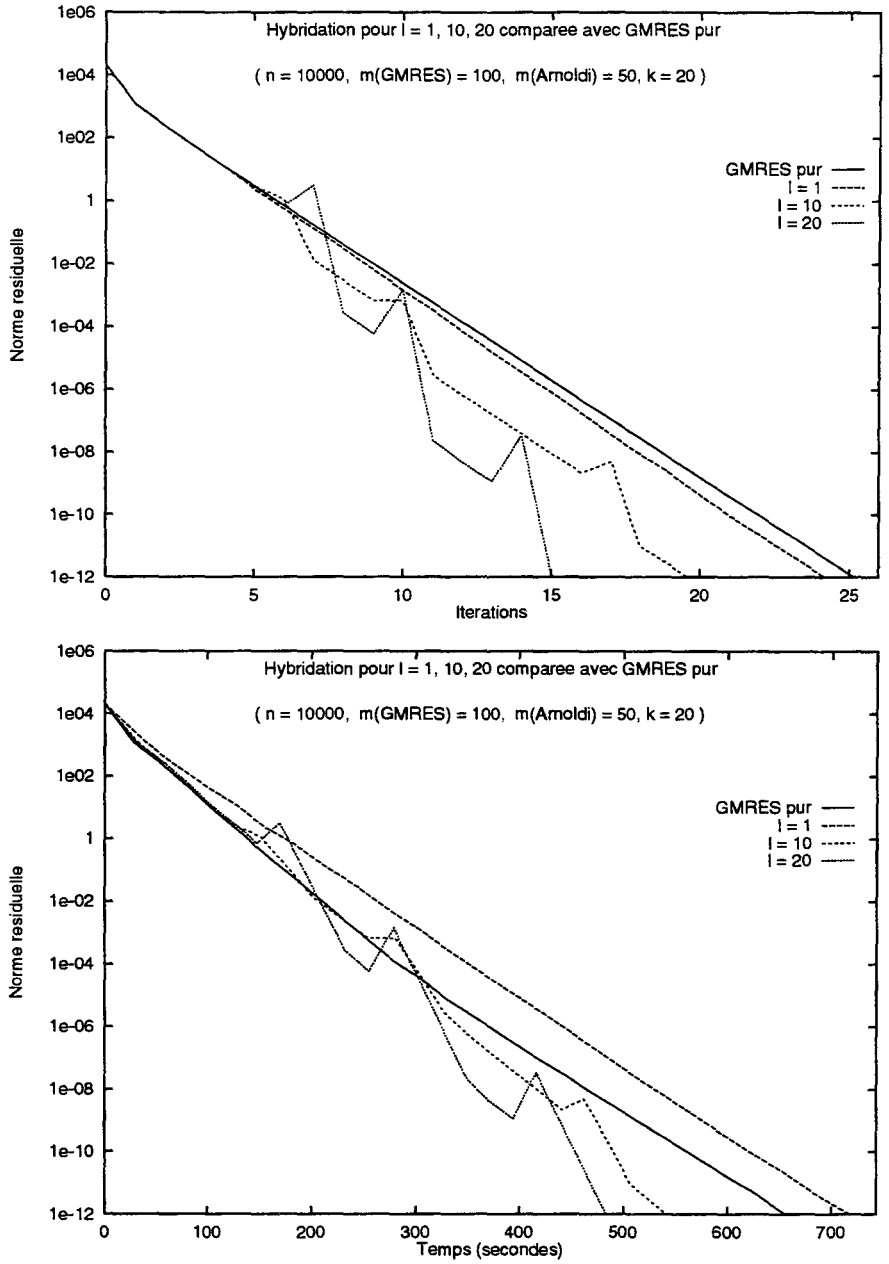


FIG. 4.8 - *Évolution de la norme résiduelle avec la méthode hybride asynchrone hétérogène comparée avec la méthode GMRES(m) pure (matrice « vp »)*

Cette étude expérimentale révèle un certain nombre de points de vue qui méritent d'être soulignés.

On peut remarquer particulièrement que, sans modifier aucun paramètre, l'accélération de la convergence peut changer d'une séance d'expérimentation à l'autre *(voir la figure 4.7)*. Le non déterminisme de l'instant du couplage hybride, dû à l'asynchronisme de

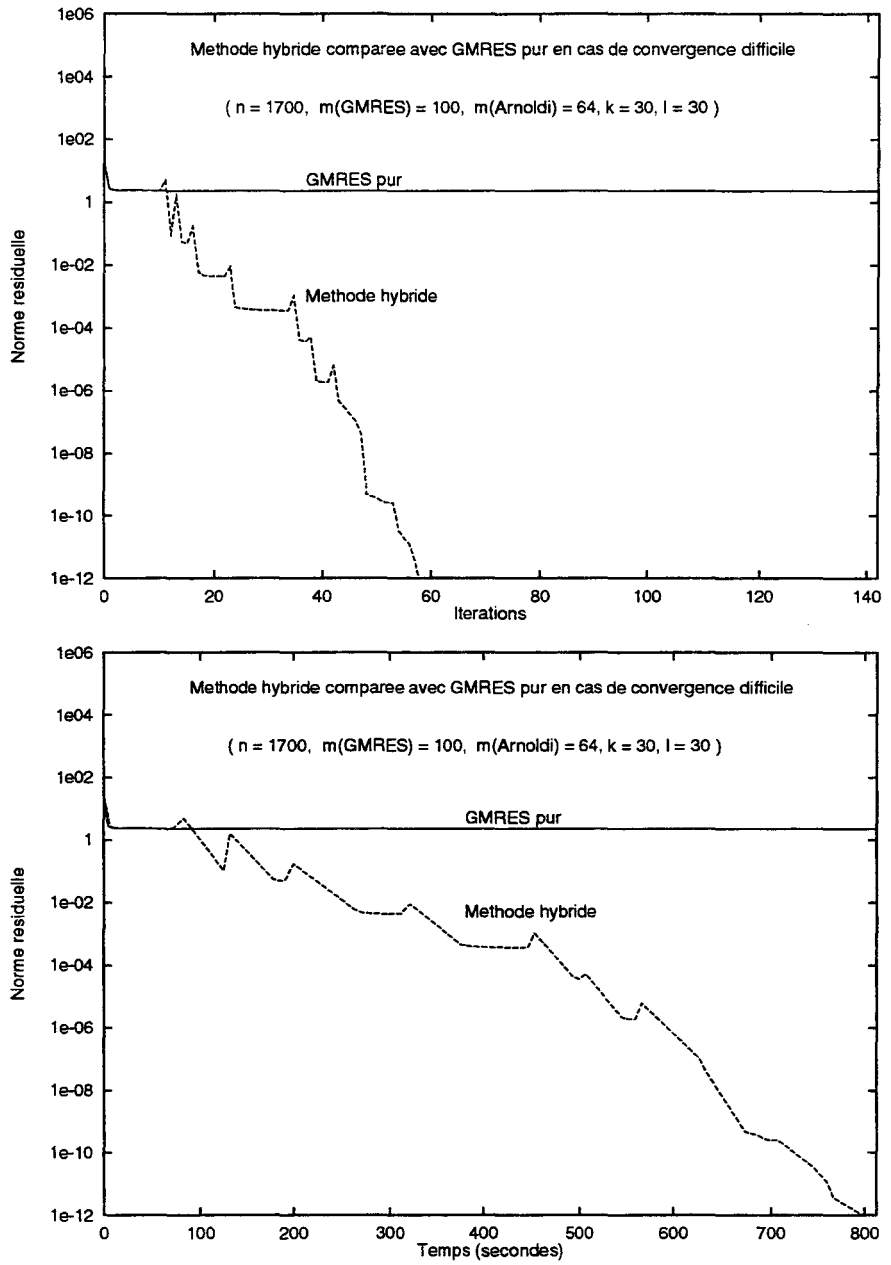


FIG. 4.9 - *Évolution de la norme résiduelle avec la méthode hybride asynchrone hétérogène comparée avec la méthode GMRES(m) pure, en cas de convergence difficile (matrice « utm1700a »)*

cette version ainsi qu'aux variations continuelles de la charge instantanée de chaque machine utilisée, explique ce phénomène. Malgré cette difficulté apparente, l'asynchronisme des processus exécutant les différentes parties de la méthode hybride évite toute synchronisation inutile et permet d'obtenir la convergence en un temps optimisé par rapport à la charge des machines. Les courbes présentées dans cette section ont chacune été relevée un

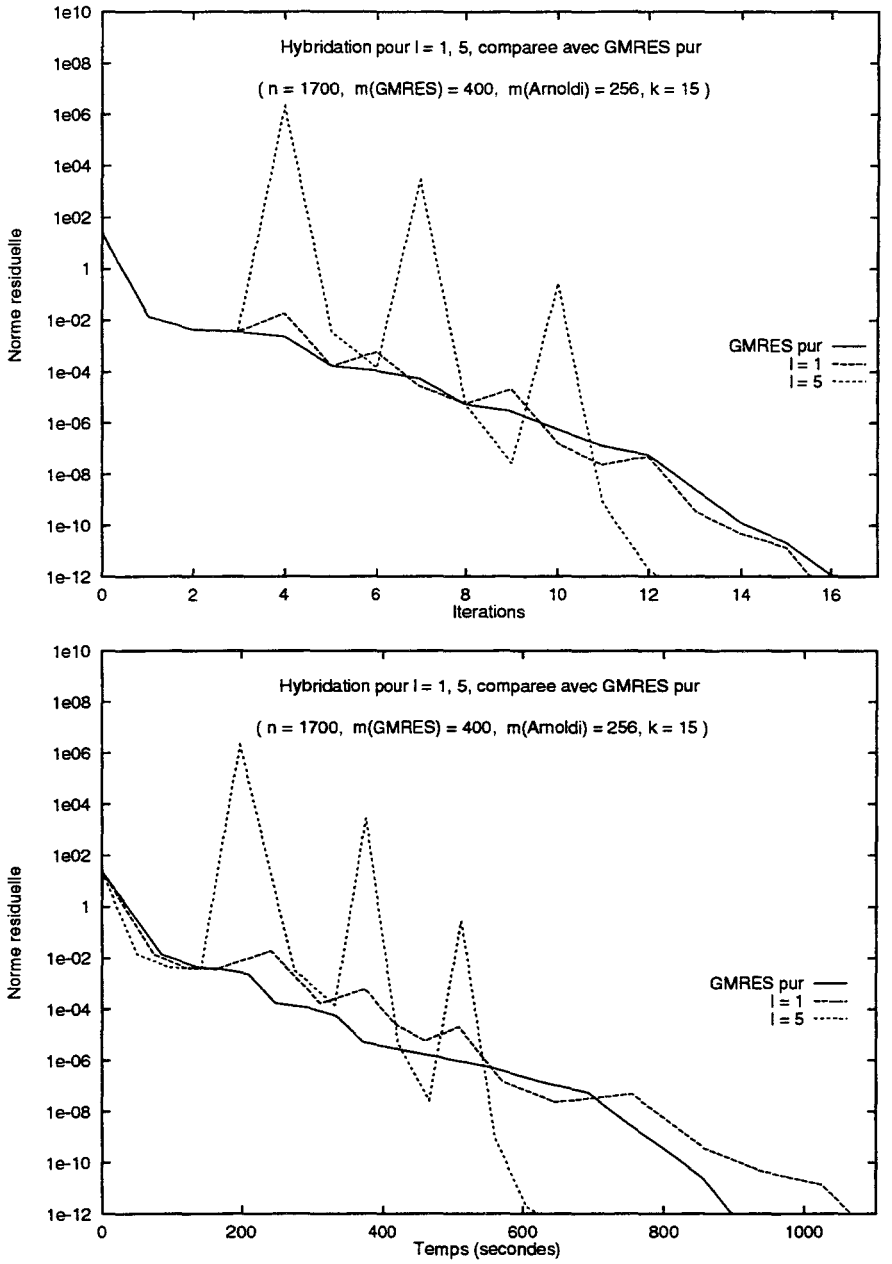


FIG. 4.10 - *Évolution de la norme résiduelle avec la méthode hybride asynchrone hétérogène comparée avec la méthode GMRES(m) pure (matrice « utm1700a »)*

jour « ordinaire », c'est à dire correspondant à une charge moyenne des machines.

Quand se produit la prise en compte des paramètres de l'hybridation par les processus GMRES(m), on remarque très souvent une augmentation temporaire, parfois très importante, de la norme résiduelle. Cependant les diminutions de cette norme résiduelle qui s'ensuivent sont bien plus rapides qu'avant la prise en compte, et globalement, malgré ces

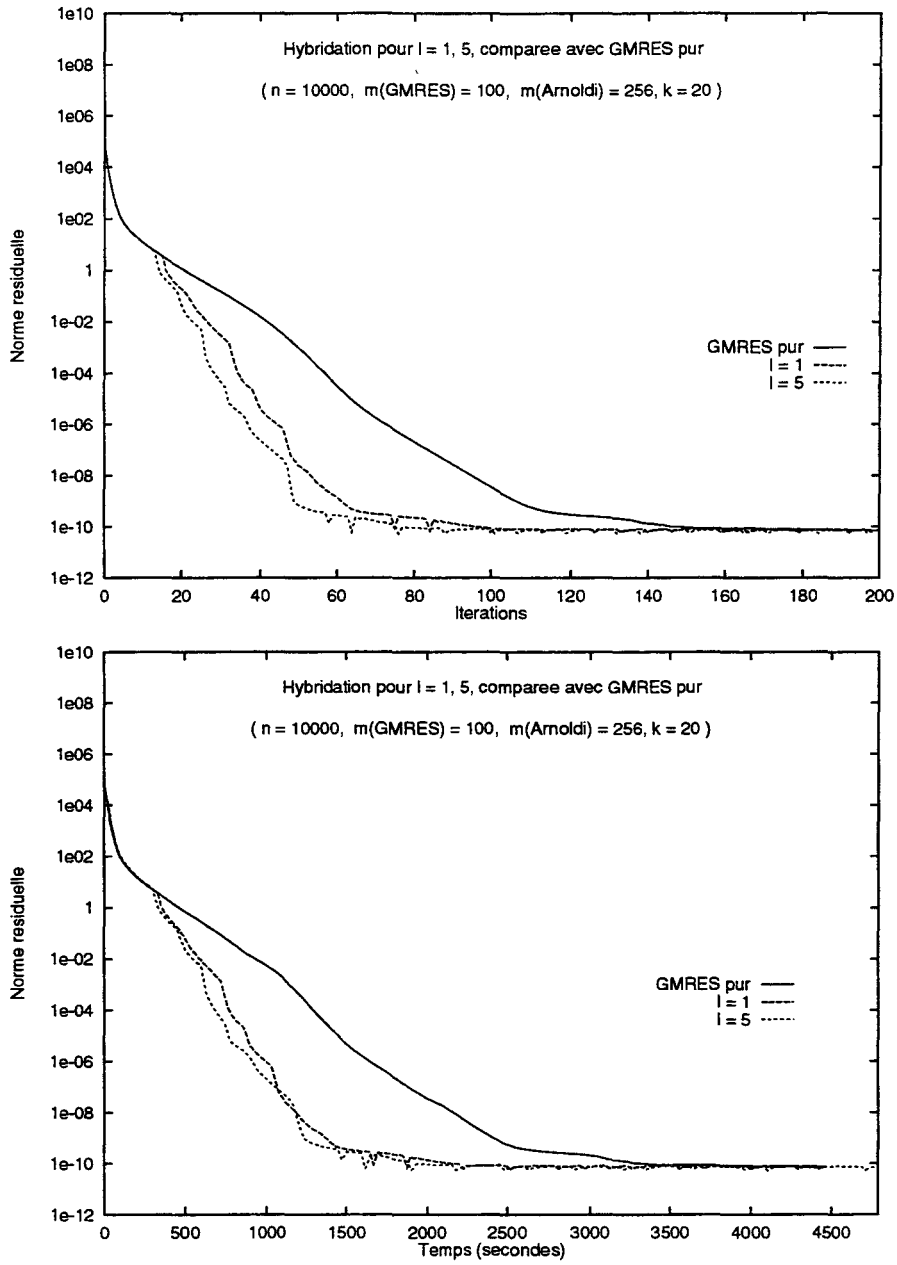


FIG. 4.11 - *Évolution de la norme résiduelle avec la méthode hybride asynchrone hétérogène comparée avec la méthode GMRES(m) pure (matrice « 5-points » avec le vecteur second membre* $b = \sum_{i=1}^{n} A_i$*)*

pics, la convergence s'accélère. Azeddine Essai, qui a développé les aspects numériques de ce travail, donne dans sa thèse [13] l'explication mathématique théorique de ce phénomène.

À cause de ces pics, de trop fréquents envois de paramètres « Least Squares » sont dommageables si on ne laisse pas chaque prise en compte avoir des répercussions sur un nombre

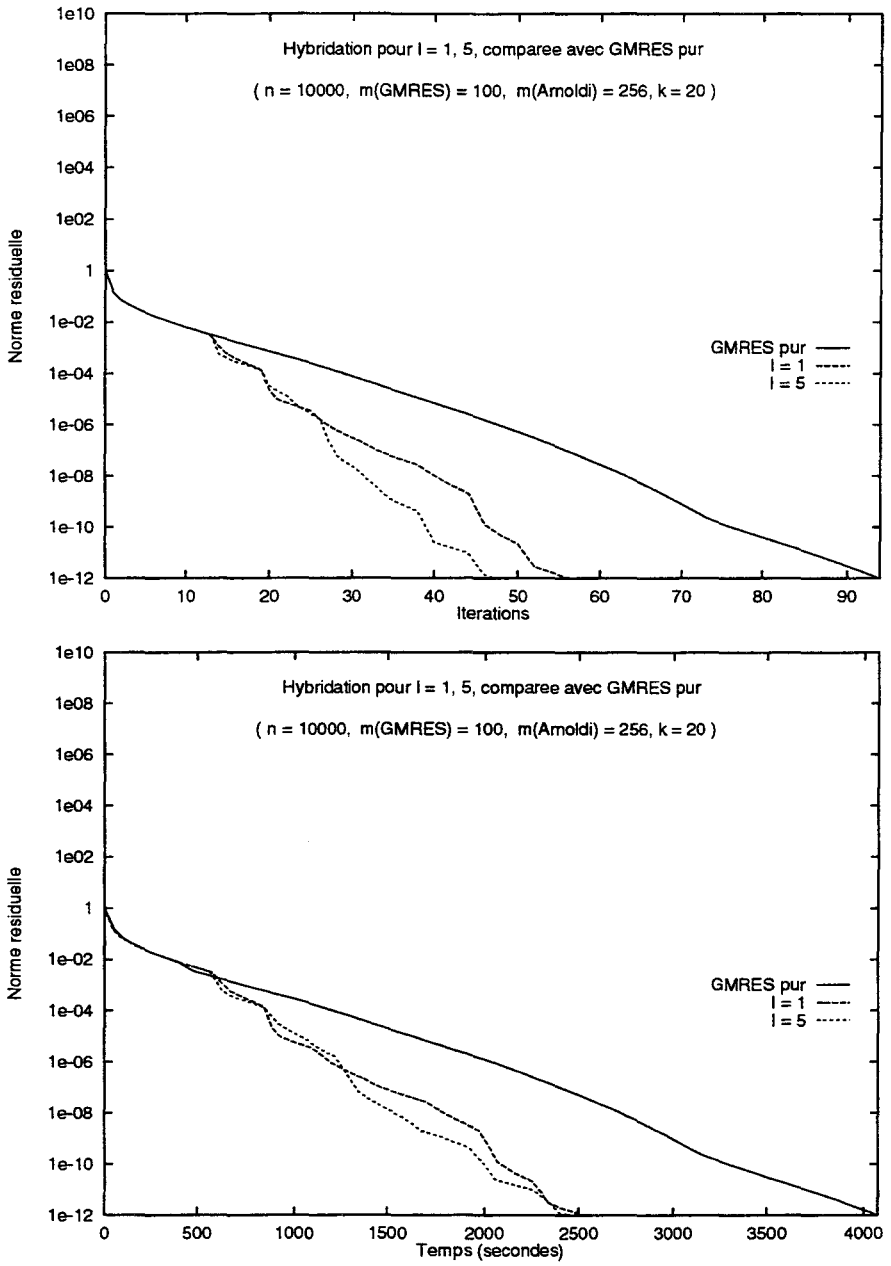


FIG. 4.12 - *Évolution de la norme résiduelle avec la méthode hybride asynchrone hétérogène comparée avec la méthode GMRES(m) pure (matrice « 5-points » avec le vecteur second membre* $b = (1, 0, \ldots, 0)^T$*)*

suffisant d'itérations GMRES(m). Si les pics sont voisins et rapprochés, on pourrait même avoir une divergence.

Un corollaire est que la machine qui exécute la méthode d'Arnoldi pour le calcul des valeurs propres n'a pas besoin d'être très performante, puisque les paramètres « Least

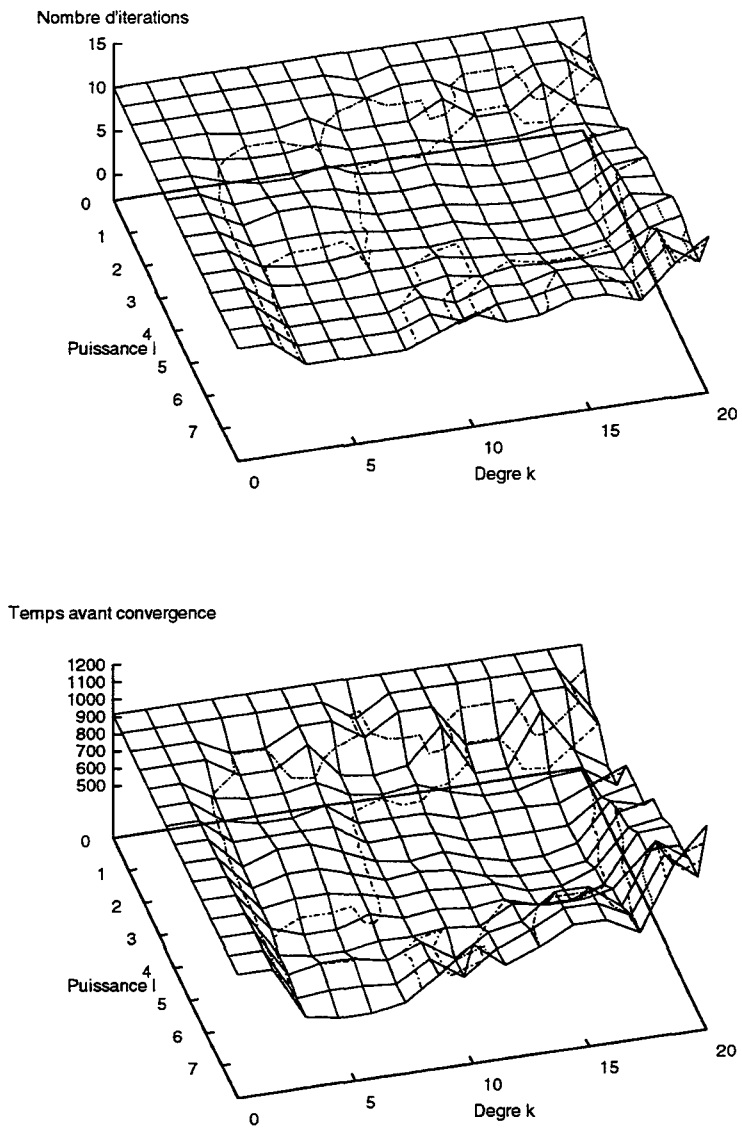


FIG. 4.13 - *Évolution du nombre d'itérations, et de la durée totale avant convergence avec la méthode hybride, en fonction du degré k du polynôme et de la puissance l (matrice « utm1700a »)*

Squares », calculés à partir des valeurs propres trouvées par la méthode d'Arnoldi, ne doivent pas être obtenus avec une trop grande fréquence. Le recours au parallélisme est cependant nécessaire, étant donné le volume des données lorsque la matrice est de taille importante. Une machine parallèle obsolète, mais d'une capacité mémoire suffisante, peut très bien convenir et permettre ainsi de réduire de manière significative le temps de calcul

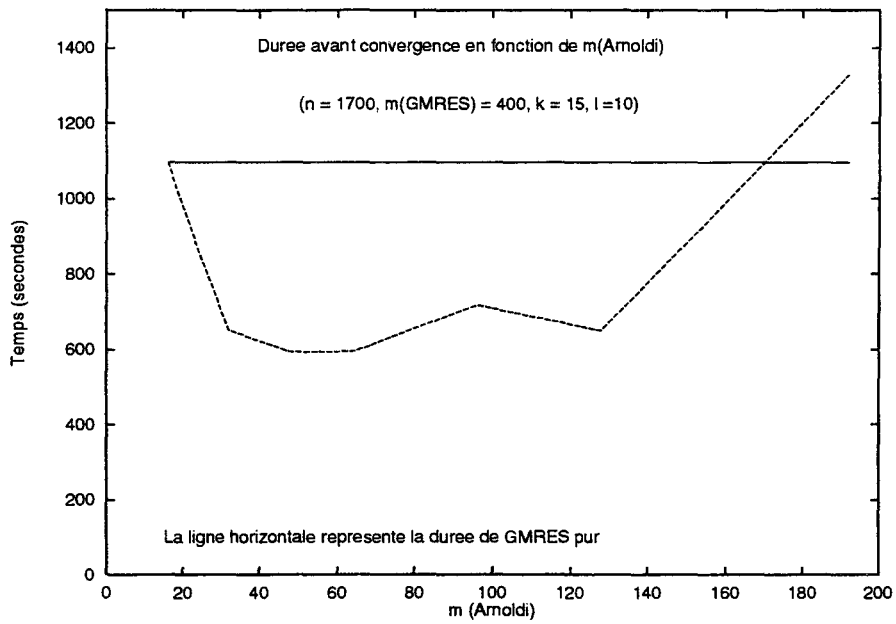


FIG. 4.14 - *Durée globale avant convergence avec l'algorithme hybride hétérogène asynchrone en fonction de la taille m(Arnoldi), comparé avec la méthode GMRES(m) pure (matrice « utm1700a »)*

GMRES(m) sur une machine performante.

4.4.3 Paramètres influant sur l'hybridation

Il est possible d'améliorer l'efficacité de l'hybridation « Least Squares » en utilisant une valeur élevée de la puissance l, c'est à dire du nombre d'itérations « Least Squares » réalisées à chaque prise en compte. L'évolution de la norme résiduelle présente alors des pics plus élevés, mais la convergence globale est encore plus rapide. Cependant, le temps de calcul de la partie parallèle de l'hybridation augmente avec cette puissance l. Ce fait est visible sur le graphe où l'on peut remarquer que la largeur des pics est plus importante sur la courbe des temps que sur celle des itérations *(voir la figure 4.10 où les échelles horizontales sont en concordance pour GMRES(m) pur)*. Pour des valeurs de l modestes, l'augmentation du temps de calcul est plus petite que le temps économisé avec l'accélération de la convergence. Cependant, pour $l = 1$, le surcoût de l'hybridation peut produire un temps plus mauvais que pour GMRES(m) pur *(voir la figure 4.8)*. Mais une augmentation excessive de l n'apporte aucune économie de temps supplémentaire, voire occasionne une perte de temps, car les pics sont très élevés et la décroissance de la norme résiduelle après de tels pics peut coûter plusieurs itérations. Avec des pics trop importants, les paramètres « Least Squares » suivants peuvent arriver avant la décroissance attendue de la norme résiduelle pour l'itération « Least Squares » précédente, et une divergence peut alors se produire.

Le degré k du polynôme « Least Squares » est aussi un paramètre important, et sa valeur doit être suffisante pour obtenir une hybridation efficace. Mais, comme pour le paramètre précédent, l'augmentation de k accroît la durée du calcul parallèle, et les mêmes phénomènes peuvent se produire.

Les expériences prouvent qu'en augmentant soit l, soit k, le nombre d'itérations diminue d'abord, puis stagne ou même augmente, et on peut trouver des valeurs optimales de k et l. Ces valeurs apparaissent aussi à propos du temps de calcul *(voir la figure 4.13)*.

4.4.4 Précision requise

Évidemment, la précision exigée pour les résultats du système linéaire est un facteur important. Mais, étant donné l'irrégularité de la décroissance de la norme résiduelle, surtout à cause de l'apparition des pics *(voir les figures 4.8, 4.9 et 4.10)*, le coût de l'accroissement de la précision exigible est difficile à estimer.

En particulier, on remarque dans la *figure 4.11* que la méthode hybride accélère beaucoup la convergence si on se contente d'une norme résiduelle de 10^{-8}, mais moins si on cherche à accroître cette précision (la convergence avec $\varepsilon_g = 10^{-12}$ n'a jamais pu être atteinte pour ce système linéaire, même avec des valeurs de k et de l élevées). La stationnarité de la norme résiduelle observée parfois dans la méthode GMRES(m) pure, à cause des erreurs d'arrondi dans le calcul, se retrouve ici aussi, dans la méthode hybride. Ainsi une toute petite augmentation de la précision requise ε_g au delà de 10^{-8} peut accroître ici considérablement le temps de calcul, voire conduire à une stagnation du résidu.

On remarque aussi sur cette figure qu'à partir du moment où la norme résiduelle est stationnaire avec la méthode GMRES(m), bien que les itérations « Least Squares », lorsqu'elles se produisent, diminuent temporairement le résidu, celui-ci, augmente de nouveau ensuite dès la reprise du calcul GMRES(m) et la méthode reste stationnaire.

Il est à souligner qu'il s'agit là d'une matrice pour laquelle l'obtention des valeurs propres est difficile : il a fallu se contenter d'un seuil de précision médiocre ($\varepsilon_a = 10^2$) pour obtenir avec la méthode d'Arnoldi un nombre de valeurs propres suffisant. Quoique la norme résiduelle diminue plus vite au début avec la méthode hybride qu'avec GMRES(m) pur, on obtient tout de même une stagnation avec ce système linéaire lorsqu'on recherche une précision importante.

L'influence de la précision requise ε_a pour l'obtention des valeurs propres est certaine, comme on pouvait le prévoir *(voir la section 4.3.3)*. Mais ses valeurs les plus efficaces dépendent grandement de la matrice utilisée. Les meilleurs résultats ont été trouvés avec $\varepsilon_a = 1$ pour la matrice « vp », $\varepsilon_a = 10^{-3}$ pour la matrice « utm1700a », et $\varepsilon_a = 10^2$ pour la matrice « 5-points ». Ces valeurs de ε_a ont ici été affinées expérimentalement, mais les travaux de A. Greenbaum [25] montrent qu'elles peuvent être estimées numériquement.

4.5 Conclusion

Les résultats expérimentaux montrent que la méthode hybride hétérogène asynchrone procure, par rapport à la méthode GMRES(m) classique, des accélérations de convergence importantes, malgré des pics momentanés, parfois très élevés, présentés par l'évolution du résidu au moment de la prise en compte des valeurs propres. Ces accélérations et ces pics sont très variables d'une matrice sur l'autre, et dépendent de nombreux paramètres dont il n'est pas facile de déterminer les valeurs les plus efficaces.

Du point de vue informatique, cette méthode permet de mieux tirer parti du parallélisme disponible, tout en conservant la granularité optimale des algorithmes impliqués. Il est ainsi possible d'utiliser plusieurs machines parallèles, chacune étant affectée à une des méthodes numériques employées, ou de partitionner une machine parallèle lorsque le degré de parallélisme optimal pour chaque algorithme numérique est inférieur à celui offert par la machine.

L'aspect hétérogène asynchrone de l'implantation réalisée est particulièrement avantageux, en permettant un déroulement indépendant et simultané de ces algorithmes dont les résultats numériques sont pris en compte sans délai d'attente, mais au prix d'une progression globale des valeurs calculées non déterministe mais convergente.

Cet aspect particulièrement souple permet d'envisager une mise en œuvre de la méthode dans des environnements parallèles de structures très différentes *(chapitre 6)*, voire en parallélisme adaptatif *(chapitre 7)*.

Chapitre 5

Algorithme parallèle entrelacé

5.1 Description

Cette autre organisation de la méthode hybride permet de n'utiliser qu'une seule machine parallèle, avec quelques machines séquentielles [3]. Il est évident que les machines parallèles n'ont pas été construites pour faire du calcul séquentiel, et elles sont souvent peu performantes dans ce type de calcul. Il est en général plus intéressant de déporter la partie séquentielle de l'algorithme sur une autre machine, même en tenant compte du surcoût des communications dû au double transfert des données. Évidemment, pendant l'exécution de cette partie séquentielle déportée, la machine parallèle attend. L'intérêt de la méthode décrite ici est de tirer parti du temps perdu en attente par la machine parallèle pendant les calculs séquentiels déportés de l'un des algorithmes, pour exécuter les calculs parallèles de l'autre, ce qui revient à entrelacer les algorithmes de base [14].

5.1.1 Organisation générale

Dans notre méthode hybride, les algorithmes GMRES/LS et Arnoldi comprennent tous deux une partie séquentielle, alors que les autres parties sont parallèles. Par conséquent, il est possible d'alterner les parties parallèles de GMRES/LS et d'Arnoldi sur la même machine, chaque algorithme utilisant le temps perdu par l'autre quand il attend les résultats de sa partie séquentielle. Un tel système entrelacé a déjà été expérimenté pour exécuter simultanément deux versions de la méthode d'Arnoldi, afin d'accélérer le calcul des valeurs propres [9, 10].

Dans notre cas, nous utilisons une machine parallèle pour dérouler l'algorithme parallèle entrelacé GMRES/LS-Arnoldi, et trois machines séquentielles : la première pour calculer les valeurs propres et les vecteurs propres de H_m pour la méthode d'Arnoldi, la deuxième pour résoudre le problème de minimisation pour la méthode GMRES(m), la troisième pour calculer les paramètres « Least Squares ». Bien que les deux calculs séquentiels des méthodes GMRES/LS et Arnoldi ne se fassent pas au même moment, l'usage de deux machines séquentielles différentes est important car il permet de recouvrir par des calculs

les temps de communications avant et après chaque partie séquentielle. En effet, la machine parallèle est ainsi disponible, dès l'envoi des données à une machine séquentielle, pour recevoir d'autres données déjà rendues disponibles par une autre machine séquentielle.

5.1.2 Entrelacement des calculs parallèles

Parmi les différents composants de la méthode hybride, c'est la méthode GMRES(m) qui est la plus importante car c'est elle qui donnera la solution du système linéaire étudié. C'est pourquoi elle sera démarrée la première dans l'algorithme entrelacé, et les paramètres devront être choisis pour minimiser ses délais d'attente.

Le processus s'exécutant sur la machine parallèle devra donc exécuter alternativement les parties parallèles des méthodes GMRES/LS et Arnoldi, et gérer les réceptions et envois de données en provenance et à destination des machines séquentielles, à chaque changement de méthode. Ce processus d'entrelacement exécutera donc l'algorithme de la *figure 5.1 (algorithme n° 5)*.

5.1.3 Caractéristiques communes de cette implantation avec la précédente

Un certain nombre de caractéristiques de cette mise en œuvre entrelacée de la méthode hybride GMRES(m)/LS(k,l)-Arnoldi sont identiques à celles de la version hétérogène asynchrone.

Ce sont:

- les différentes étapes du calcul *(voir la section 4.1.1 page 53)*,
- le principe de l'accumulation des valeurs propres (*voir la section 4.1.2 page 54*),
- tous les paramètres liés aux méthodes numériques (n, m(GMRES), m(Arnoldi), k, l) et les paramètres d'implantation c, ε_g, ε_a, **ITERMIN**, **RMIN**, **NVPMIN** *(voir la figure 4.4 page 59)*. Il est à noter cependant que des contraintes supplémentaires existent entre certains de ces paramètres, afin de ne pas être pénalisé par les délais d'attente dûs à l'entrelacement *(voir les sections 5.3 page 80 et 5.4.3 page 84)*,
- Les choix concernant l'architecture et le langage *(voir les sections 4.2.1 page 54 et 6.6 page 105)*.

ALGORITHME 5 : ENTRELACEMENT GMRES/LS-ARNOLDI

1. Initialiser les méthodes GMRES(m)/LS(k,l) et Arnoldi.
2. **Méthode GMRES(m) :**
   (a) Exécuter l'étape n° 2 de l'*algorithme 2 (voir page 28).*
   (b) Envoyer les données à la machine séquentielle n° 1 pour le calcul de la solution approchée (étape n° 3 de l'*algorithme 2).*
3. **Méthode d'Arnoldi :**
   Si l'itération courante n'est pas la première alors
   (a) Attendre la réception des valeurs propres et des vecteurs propres calculés par la machine séquentielle n° 2.
   (b) Exécuter les étapes n^{os} 4 à 6 de l'*algorithme 3 (voir page 31).*
   (c) Envoyer les valeurs propres à la machine séquentielle n° 3 pour le calcul des paramètres « Least Squares ».

   fin si
4. **Méthode d'Arnoldi :**
   (a) Faire un redémarrage de la méthode d'Arnoldi et exécuter l'étape n° 2 de l'*algorithme 3.*
   (b) Envoyer les données à la machine séquentielle n° 2 pour le calcul des valeurs propres et des vecteurs propres.
5. **Méthode GMRES(m) :**
   (a) Attendre la réception de la solution approchée calculée par la machine séquentielle n° 1.
   (b) Faire le calcul du résidu et le test de l'étape n° 4 de l'*algorithme 2.*
   (c) Si un redémarrage est nécessaire alors
   Si des paramètres « Least Squares » sont disponibles alors
   Méthode « Least Squares » :
   Recevoir les paramètres « L. S. ».
   Faire l itérations « L. S. » (étape n° 4 de l'*algorithme 4, voir page 33).*
   Si un redémarrage est nécessaire alors
   Aller à l'étape n° 2 du présent algorithme.
   fin si
   sinon
   Aller à l'étape n° 2 du présent algorithme.
   fin si

FIG. 5.1 - *Algorithme n° 5 : La méthode hybride entrelacée GMRES/LS-Arnoldi*

5.2 Choix d'implantation

5.2.1 Architecture parallèle utilisée

Cette version a été mise en œuvre en utilisant le réseau de l'ETCA [1], qui a offert à notre équipe l'accès à son réseau comprenant notamment une Connection Machine CM5

1. Établissement Technique Central de l'Armement, 16 bis avenue Prieur de la côte d'Or, 94114 Arcueil Cedex, France

de 32 nœuds et plusieurs stations de travail Sun.

La CM5 va donc gérer la partie parallèle entrelacée de la méthode hybride, alors que trois stations Sun exécuteront les parties séquentielles. Ces stations de travail ont été préférées à l'ordinateur frontal de la Connection Machine car elles sont plus performantes, même en tenant compte du surcoût dû aux communications engendrées par le transfert des données sur une autre machine.

La CM5

La CM5 est une machine parallèle organisée en nœuds de calcul (32 pour la machine de l'ETCA). Chacun de ces nœuds comprend une mémoire de 32 Mo, un processeur de calcul sur 64 bits et plusieurs processeurs de contrôles, chargés de gérer l'interfaçage avec le réseau, les entrées-sorties et les diagnostics d'erreurs. Le réseau de communications est triple : en plus du réseau de données, on trouve un réseau de contrôle et un réseau de diagnostic.

Cette machine peut-être utilisée aussi bien en parallélisme de données qu'en parallélisme de tâches, le système de contrôle permettant de faire fonctionner les processeurs élémentaires en mode SIMD.

Pour les raisons exposées dans la *section 4.2.1* c'est dans ce mode que la version entrelacée décrite dans ce chapitre utilisera la CM5. De plus, le compilateur « CMFortran », version de Fortran 90 adapté à la CM5, permet justement une programmation en parallélisme de données. Il est à noter d'ailleurs qu'entre les versions MasPar et CM5 de la méthode d'Arnoldi, seules les directives de placement des données ont dû être modifiées. En effet, MPFortran et CMFortran sont deux versions de Fortran 90 adaptées respectivement à la MasPar et à la CM5, dans lesquelles les particularités de placement liées à la machine sont gérées par des directives placées dans des lignes de commentaires (commençant respectivement par CMPF et CMF$).

5.2.2 Gestion des communications

Les communications entre les processeurs de la CM5 étant gérées implicitement de manière optimisée par les bibliothèques du compilateur CMFortran, seules les communications avec les machines séquentielles doivent être gérées de manière explicite. Comme pour la version précédente, les bibliothèques PVM ont été utilisées. Il est à noter que pour toutes ces communications entre la CM5 et une station Sun, il s'agit dans chaque cas (étapes n^{os} 2b, 5a, 4b, 3a, 3c et 5c de l'algorithme 5 page 77) d'un volume de données restreint.

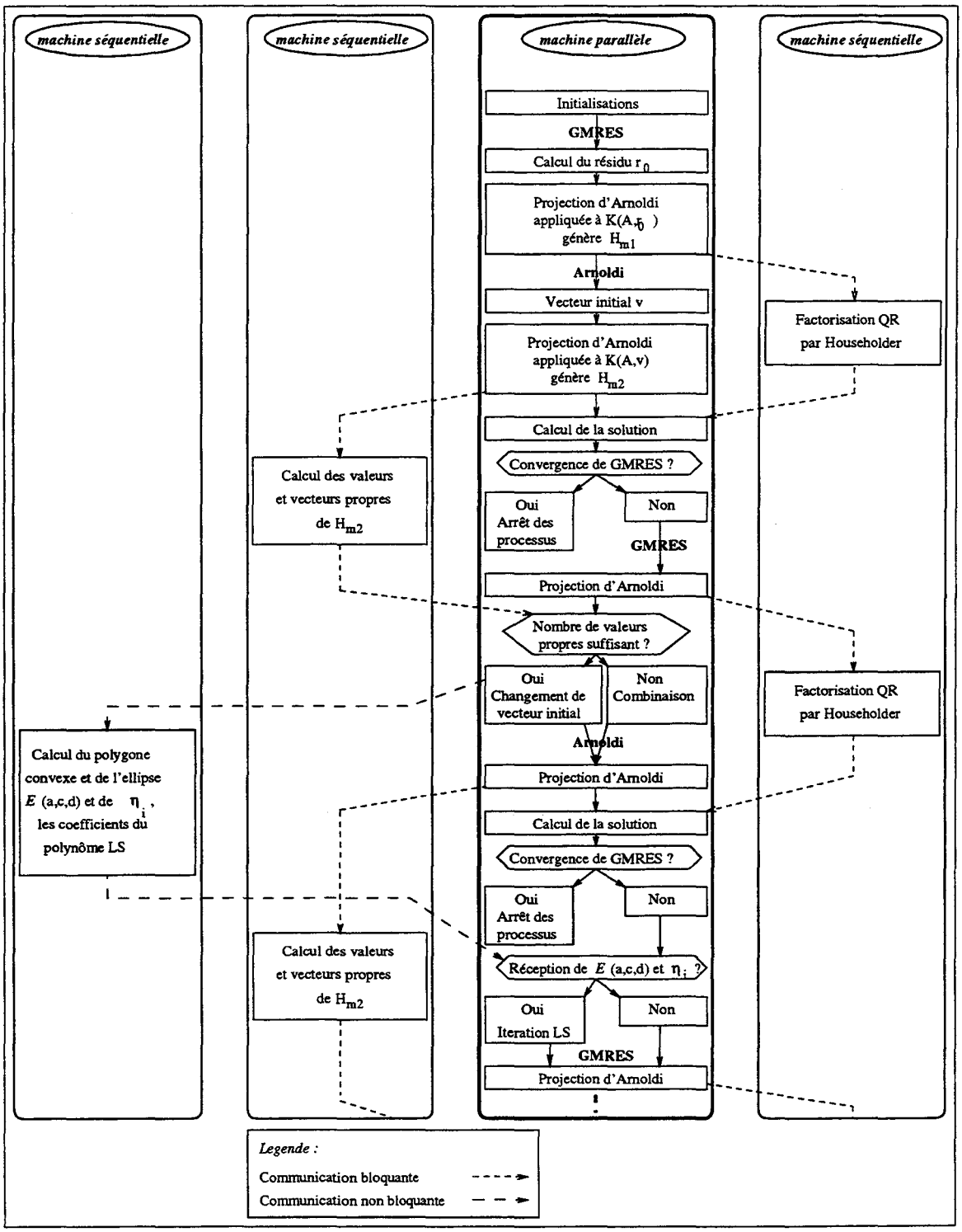


FIG. 5.2 - *Schéma de principe de la méthode hybride GMRES/Least Squares-Arnoldi parallèle entrelacée*

5.2.3 Mise en place de la matrice

Comme pour la version hétérogène asynchrone, le problème de stockage de la matrice s'est posé, et la méthode a d'abord été testée avec des matrices générées dynamiquement à chaque exécution. Puis quelques matrices industrielles issues de MatrixMarket ont aussi été stockées sur disque et testées *(voir page 63)*.

La mise en place des matrices pour la méthode entrelacée s'avère cependant plus aisée : les processus parallèles exécutant les méthode GMRES/LS et Arnoldi se trouvent en effet sur la même machine, et constituent en fait un seul et même processus déroulant l'algorithme 5 *(décrit page 77)*. La matrice n'a donc besoin d'être accessible que par ce seul algorithme, et sa mise en place s'en trouve facilitée et accélérée (pas d'accès concurrents à un même fichier de données). De plus, aucune des méthodes numériques employées ne modifiant les éléments de la matrice, aucun effet de bord n'est à craindre en passant d'un composant à l'autre de l'algorithme entrelacé.

5.3 Asynchronisme déterministe des réceptions

Par comparaison avec la méthode GMRES/LS-Arnoldi hétérogène asynchrone parallèle, la principale caractéristique distinctive de la présente méthode est l'existence de points de synchronisation entre les parties parallèles des algorithmes GMRES/LS et Arnoldi, à chaque étape de l'entrelacement : si, par exemple, la partie séquentielle de la méthode GMRES/LS est terminée alors que la partie parallèle de la méthode d'Arnoldi est encore en cours, GMRES/LS devra attendre que la machine parallèle soit libre pour poursuivre le calcul. Il se peut aussi qu'une partie parallèle se termine, et que celle de l'autre algorithme qui doit se mettre en place ensuite attende après les résultats d'un calcul séquentiel *(ces différents cas se retrouvent sur la figure 5.3)*. Par contre, les envois de valeurs propres au processus de calcul des paramètres « Least Squares », ainsi que la réception de ces paramètres par le processus GMRES/LS restent naturellement asynchrones, comme dans la version du *chapitre 4*.

Pour pallier aux inconvénients de ces synchronisations supplémentaires, certains paramètres devront être ajustés afin d'éviter des pertes de temps quand l'un des algorithmes entrelacés attend l'autre. Il s'agit des paramètres qui affectent la durée des calculs dans l'un ou l'autre des algorithmes entrelacés, et dont l'influence a déjà été explicitée dans la *section 4.3.3*.

On peut notamment jouer sur les tailles des deux sous-espaces de Krylov des méthodes GMRES(m) et Arnoldi pour minimiser les temps d'attente : la durée d'une itération de chacune de ces méthodes dépend en effet étroitement de la valeur de m la concernant. Pour ce réglage, il faut se souvenir que la tâche prioritaire est GMRES(m)/LS, et il est important de s'assurer qu'elle ne subira aucun délai d'attente. Cette condition sera réalisée

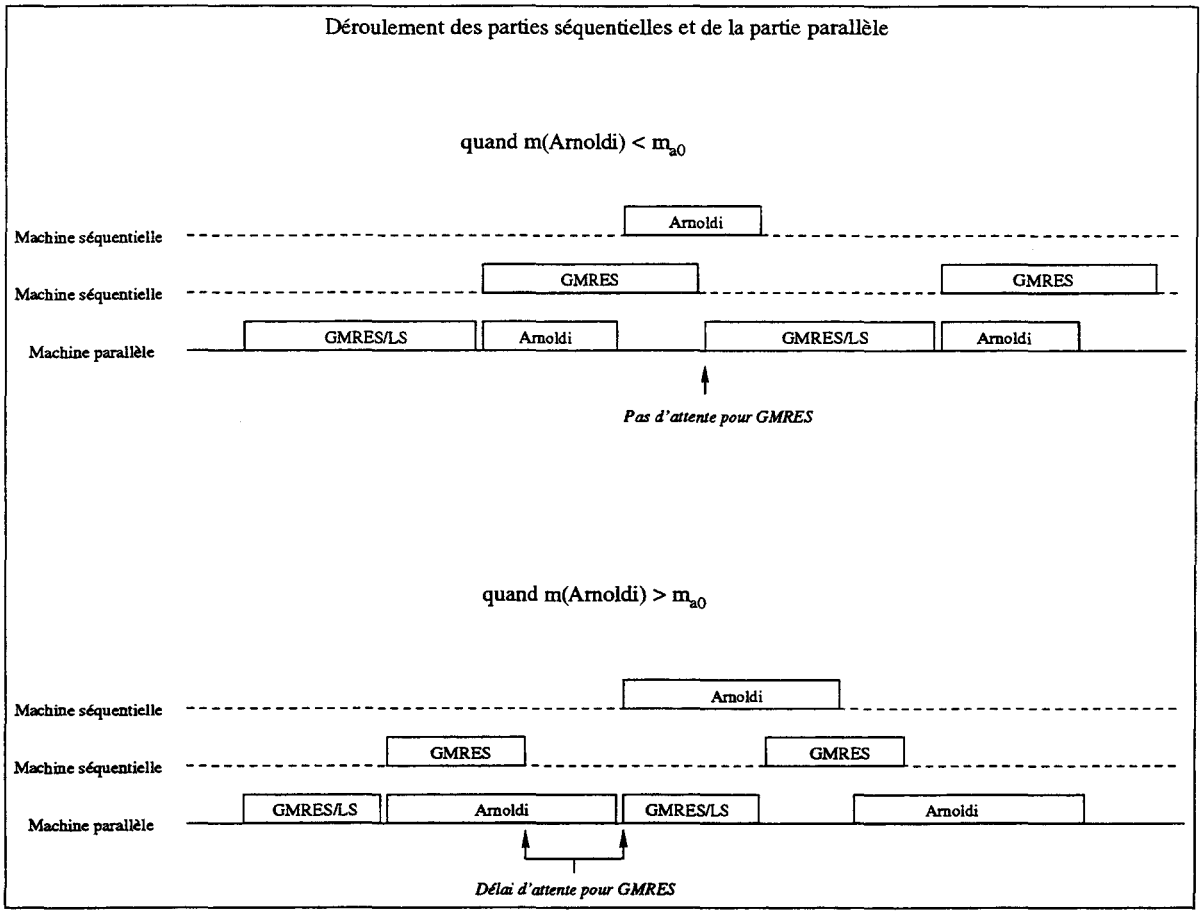


FIG. 5.3 - *Représentation des attentes aux points de synchronisation de la méthode entrelacée, en fonctions des valeurs relatives de m(GMRES) et m(Arnoldi)*

quand la durée t_a de la partie parallèle de la méthode d'Arnoldi sera inférieure à la durée t_g de la partie séquentielle de la méthode GMRES(m). Ces deux durées dépendent des complexités respectives des deux algorithmes, étroitement liées aux valeurs de m(GMRES) et de m(Arnoldi). Mais elles dépendent aussi des vitesses de calcul des machines qui les hébergent, ainsi que des charges de ces machines. Pour une valeur donnée de m(GMRES), et pour une charge donnée des deux machines, on peut trouver la valeur m_{a0} de m(Arnoldi) telle que $t_a = t_g$. Pour éviter tout délai d'attente de GMRES(m)/LS, il paraît judicieux de choisir m(Arnoldi) $< m_{a0}$, ainsi les délais d'attente dûs à la synchronisation se trouveront tous subis par le processus d'Arnoldi *(voir la figure 5.3)*. Cependant, cette contrainte supplémentaire sur les valeurs de m peut être gênante: si m(Arnoldi) est trop petit, on obtiendra trop peu de valeurs propres pour réaliser une hybridation efficace; si m(GMRES) est trop grand (dans le but d'avoir une valeur plus grande de m_{a0}), les calculs de la méthode GMRES(m) deviendront trop lourds, ralentissant la résolution du système. De plus, le choix de ces deux valeurs est très dépendant de la matrice étudiée. Lors des tests, la valeur de m(GMRES) a d'abord été fixée, afin d'obtenir des performances satisfaisantes avec la

méthode GMRES(m) pure. La même valeur de ce paramétre a été utilisée avec la méthode entrelacée, en employant différentes valeurs de m(Arnoldi) dont l'influence est reflétée par la *figure 5.8* et discutée dans la *section 5.4.3.*

5.4 Résultats numériques

5.4.1 Exemples choisis pour les tests

Afin de retrouver les mêmes conditions expérimentales dans cette implantation entrelacée que dans la précédente hétérogène asynchrone, les mêmes matrices ont été testées *(voir page 63)*. Pour la matrice « 5-points », comme dans l'implantation précédente, un second vecteur second membre a été testés, le premier conduisant à une stagnation du résidu GMRES/LS.

5.4.2 Évolution de la norme résiduelle

On retrouve encore sur les courbes de la norme résiduelle les mêmes pics qui avaient été remarqués avec la méthode GMRES/LS-Arnoldi parallèle hétérogène asynchrone. Mais il y a une différence importante : à cause des synchronisations supplémentaires, la largeur des pics est constante tout au long de chaque courbe des temps, et cette largeur n'est pas modifiée en fonction de l, malgré les variations de la durée des calculs induites par les modifications de ce paramètre *(voir les figures 5.4, 5.5 et 5.7)*. Ce fait met en évidence l'existence de pertes de temps dues aux délais d'attentes induits à chaque changement d'algorithme de l'entrelacement. Malgré les précautions prises pour minimiser les effets de ces synchronisation pour l'algorithme prioritaire (GMRES/LS), celles-ci restent donc pénalisantes.

Cependant, la méthode hybride permet tout de même un gain de temps important par rapport à la méthode GMRES(m) pure, tout au moins quand la convergence peut être obtenue sans stagnation du résidu. Ce cas, observé pour la matrice « 5-points » avec le vecteur second membre $b = \sum_{i=1}^{n} A_i$, comme pour la version hétérogène asynchrone, montre cependant une accélération de la convergence si on se contente d'une précision plus modeste (10^{-8} par exemple) pour le résidu *(voir la figure 5.6)*.

De plus, on peut remarquer que le temps utilisé pour exécuter la méthode d'Arnoldi sur la Connection Machine selon la méthode hybride est « gratuit » (tout au moins en raisonnant en mode de fonctionnement dédié). En effet le temps qui était perdu de toute façon, sur la machine parallèle dans la méthode GMRES(m) pure, pour attendre la résolution du problème d'optimisation *(étape n° 3 de l'algorithme 2 page 28)*, est maintenant mis à profit pour exécuter la partie parallèle de la méthode d'Arnoldi. Le coût supplémentaire de la méthode hybride provient des parties séquentielles déportées (calcul des valeurs et vecteurs propres de H_m dans la méthode d'Arnoldi et calcul des paramètres de la méthode

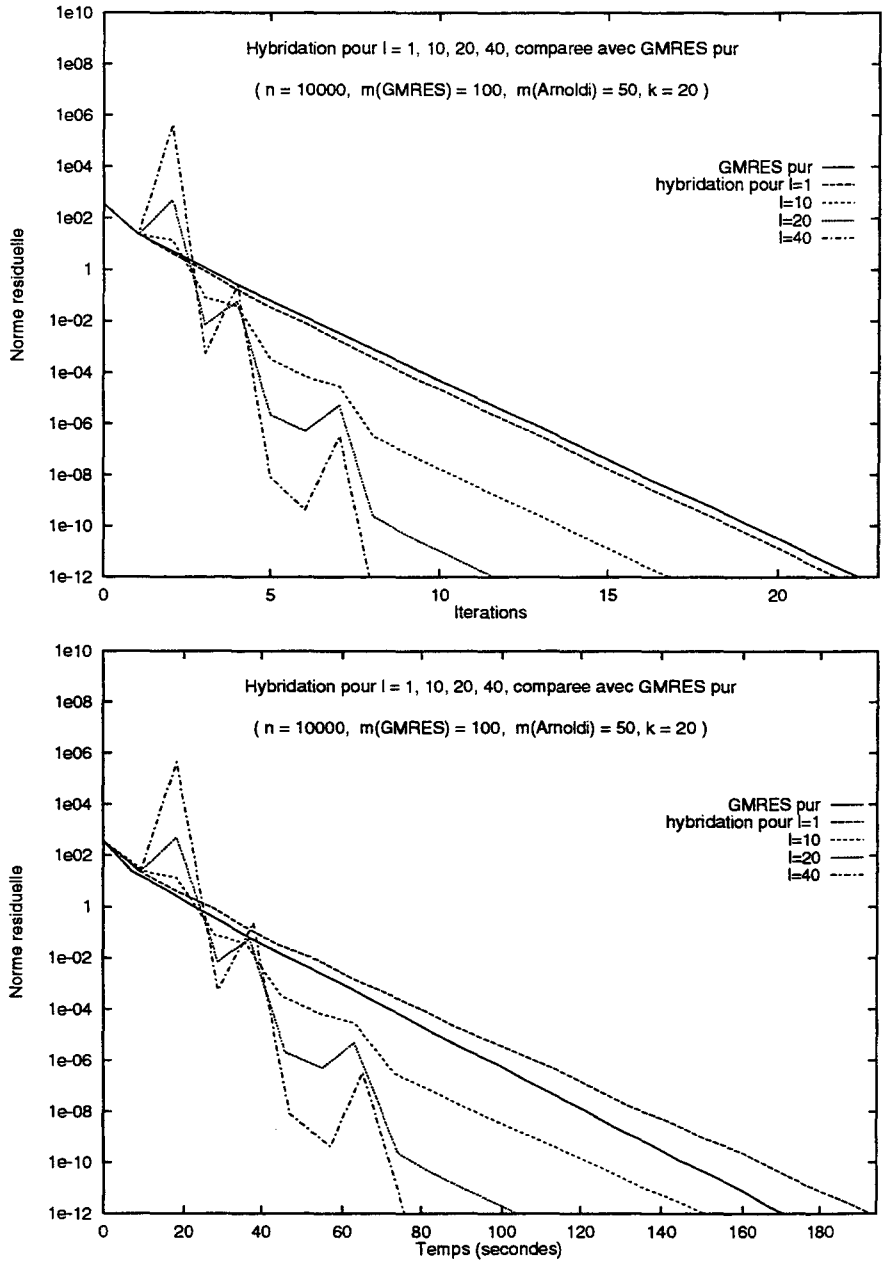


FIG. 5.4 - *Évolution de la norme résiduelle avec l'algorithme hybride entrelacé comparé avec la méthode GMRES(m) pure (matrice « vp »)*

« Least Squares »). Ces calculs consomment des ressources, mais s'exécute en simultanéité avec les autres calculs parallèles, ce qui n'occasionne pas de durée supplémentaire pour la résolution du système linéaire.

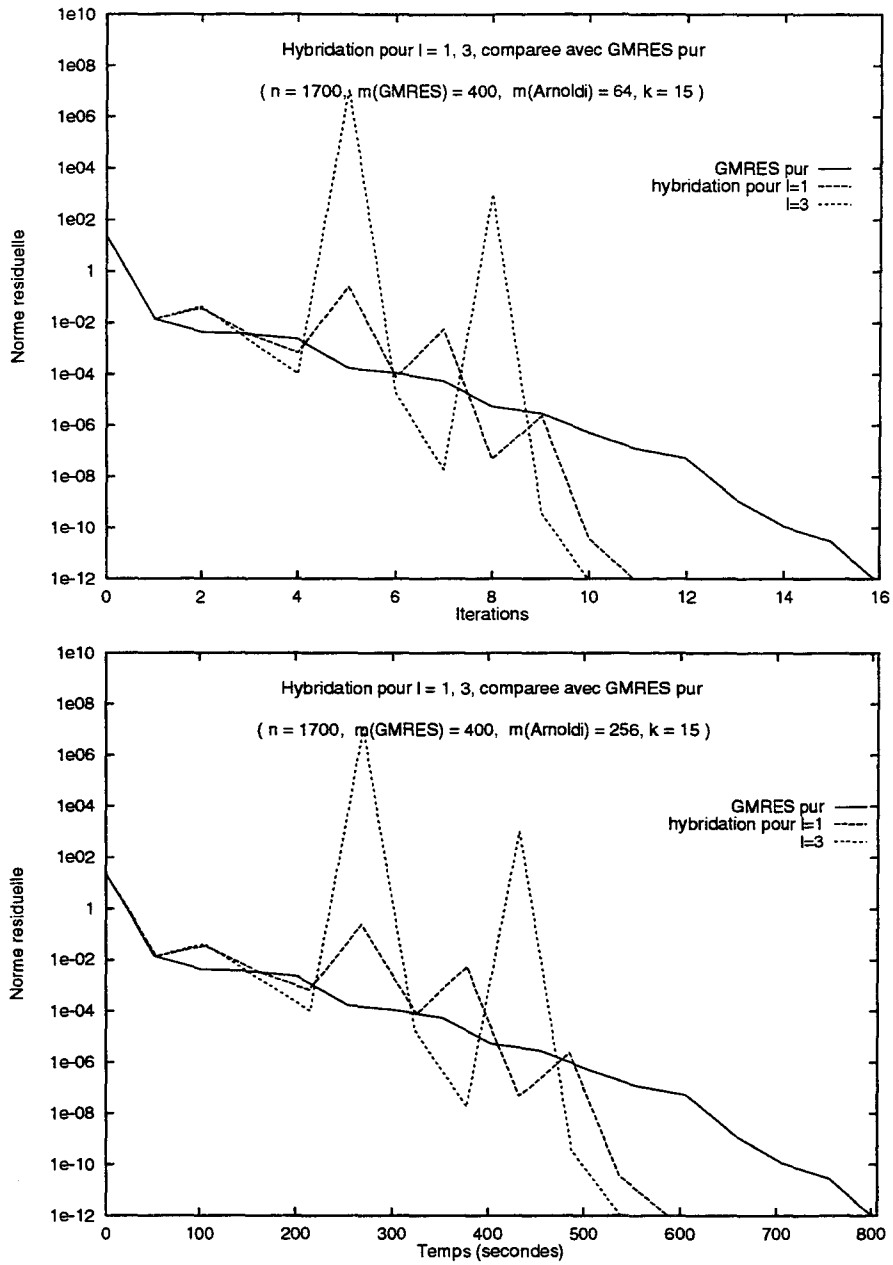


FIG. 5.5 - *Évolution de la norme résiduelle avec l'algorithme hybride entrelacé comparé avec la méthode GMRES(m) pure (matrice « utm1700a »)*

5.4.3 Taille de l'espace de projection

Quand on étudie le temps global nécessaire à la convergence de la méthode hybride en fonction de la taille m du sous-espace de Krylov de l'algorithme d'Arnoldi *(voir la figure 5.8)*, on observe l'existence d'une valeur optimale pour m. En dessous de cet optimum, m n'est pas suffisant pour calculer assez de valeurs propres pour un calcul hybride efficace,

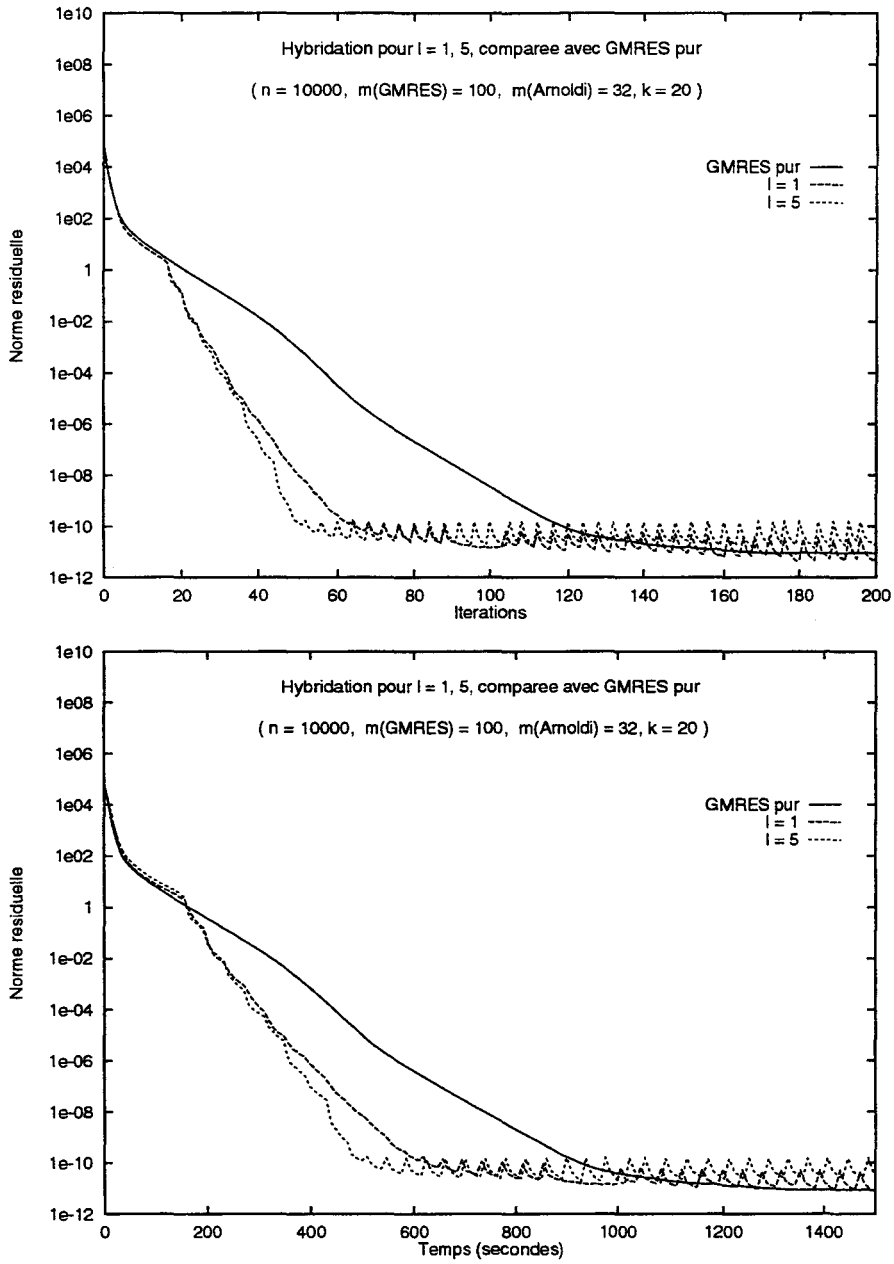


FIG. 5.6 - *Évolution de la norme résiduelle avec l'algorithme hybride entrelacé comparé avec la méthode GMRES(m) pure (matrice « 5-points » avec le vecteur second membre* $b = \sum_{i=1}^{n} A_i$*)*

et les performances de la méthode hybride tendent vers celles de la méthode GMRES(m) pure, voire pire à cause coût de l'algorithme d'entrelacement. Au-dessus de cet optimum, les calculs concernant l'algorithme d'Arnoldi deviennent trop lents par rapport à ceux concernant GMRES(m). Les mêmes observations ont pu être faites avec la méthode hétérogène asynchrone *(voir la figure 4.14)*, mais ici, l'allongement des calculs produit par

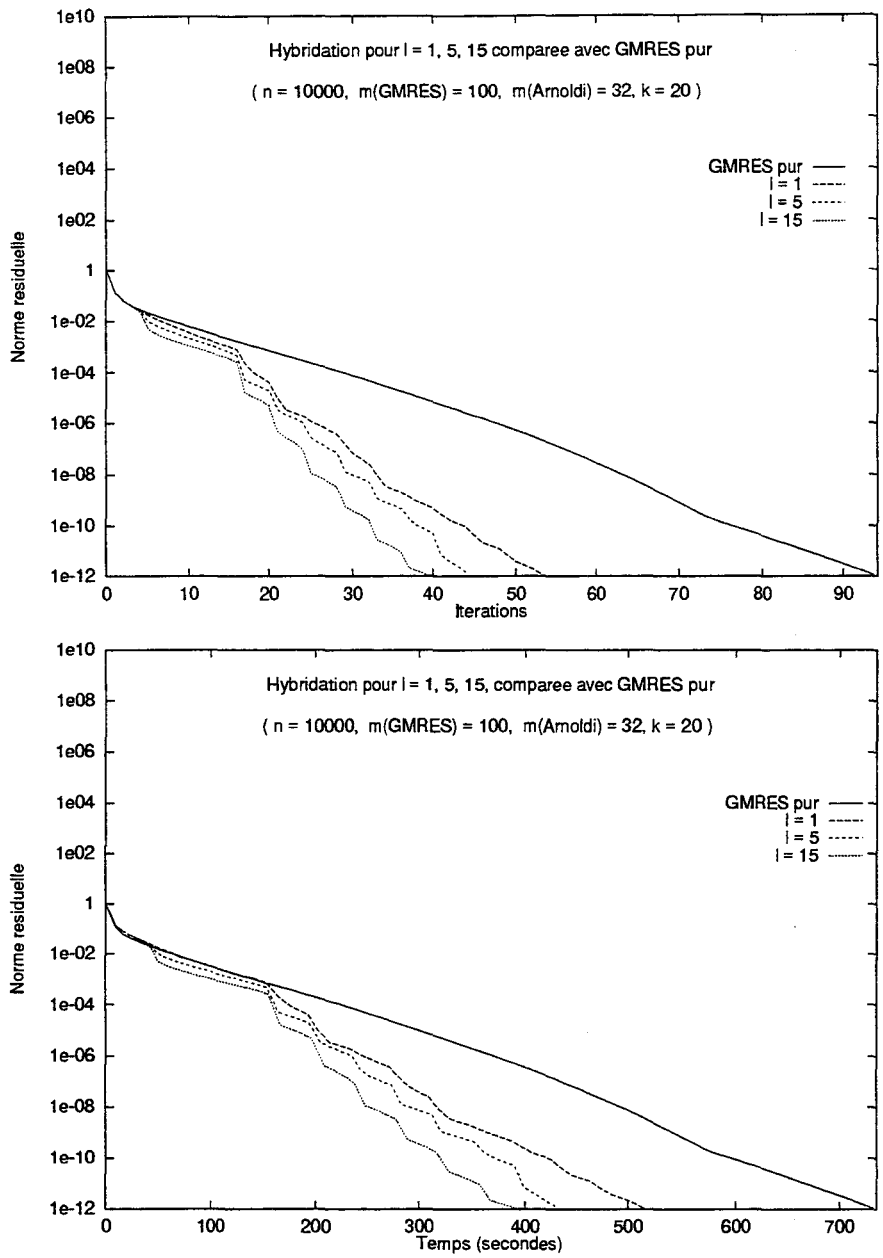


FIG. 5.7 - *Évolution de la norme résiduelle avec l'algorithme hybride entrelacé comparé avec la méthode GMRES(m) pure (matrice « 5-points » avec le vecteur second membre* $b = (1, 0, \ldots, 0)^T$*)*

l'augmentation de m(Arnoldi) est plus critique à cause de l'entrelacement. La projection d'Arnoldi sur la CM5 dure alors plus longtemps que la partie séquentielle de GMRES(m) qu'elle est censée recouvrir et des délais d'attente apparaissent dans la tâche prioritaire GMRES(m), ce qui détériore ses performances. Ces délais d'attente peuvent être extrêmement pénalisants.

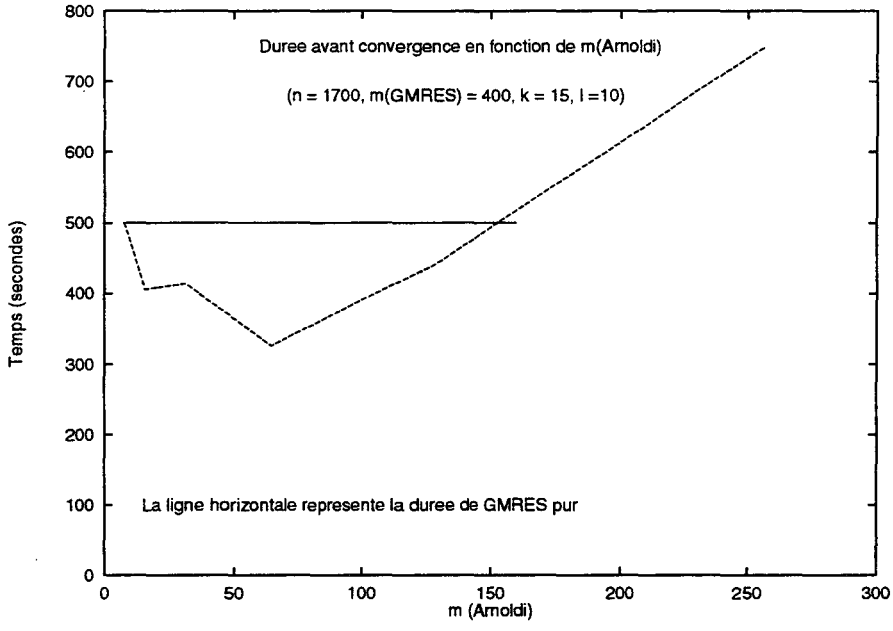


FIG. 5.8 - *Évolution de la durée globale avant convergence avec l'algorithme hybride entrelacé en fonction de la taille m(Arnoldi), comparé avec la méthode GMRES(m) pure (matrice « utm1700a »)*

La méthode hybride entrelacée nécessite donc un ajustement délicat de ce paramètre pour obtenir les meilleures performances. Néanmoins, elle reste plus rapide que la méthode GMRES(m) pure dans un important domaine de valeurs de m.

5.5 Conclusion

La méthode hybride entrelacée permet un accroissement significatif de la convergence par rapport à la méthode GMRES(m) classique. Mais l'accélération obtenue dépend énormément de la matrice utilisée, et aussi de nombreux paramètres dont les valeurs optimales ne sont pas toujours faciles à trouver. Cette difficulté de mise au point ne doit pas rebuter, tant il est certain que des méthodes aussi complexes en environnement parallèle hétérogène ne peuvent pas être des « boîtes noires » utilisables sur n'importe quelle matrice sans ajustement.

Comparativement à la méthode hétérogène asynchrone décrite au *chapitre 4*, la présente méthode présente, et c'était prévisible, moins de souplesse à cause des synchronisations induites par l'entrelacement, et l'accélération de la convergence est moins importante à cause de ces synchronisations supplémentaires. De plus, l'ajustement des paramètres, afin d'obtenir des délais d'attente minimaux aux points de synchronisation, s'avère délicat mais déterminant. Par contre, les besoins en ressources mémoire sont réduits du fait que la ma-

trice n'est présente qu'une seule fois pour les deux algorithmes qui l'utilisent. C'est l'un des intérêts majeurs de cette méthode, qui évite aussi, par la même occasion un double temps de chargement des données et des communications d'une machine parallèle à l'autre. De plus, l'utilisation d'une seule machine parallèle réduit le coût d'exploitation de la méthode hybride.

L'originalité de la présente méthode réside dans le recouvrement par des calculs des délais d'attente induits par le report des calculs séquentiels sur une autre machine, report qui est indispensable pour beaucoup de machines parallèles dont les performances sont médiocres en termes d'exécution séquentielle. Il est à remarquer que certains calculs traitées en mode séquentiel dans cette version (il s'agit du calcul des valeurs propres et de la factorisation QR) auraient pu de façon avantageuse être confiées à une machine vectorielle, si nous avions disposé d'une telle machine. Cela aurait notamment pu réduire les attentes aux points de synchronisation lorsque le calcul séquentiel est pénalisant.

Ce type d'implantation prendra donc tout son intérêt sur un site qui ne dispose que d'une seule machine parallèle, et notamment si les performances de celle-ci sont fortement dégradées en mode séquentiel ou si une machine vectorielle permet d'accélérer les calculs déportés.

Chapitre 6

Construction modulaire d'applications parallèles hétérogènes

Lorsqu'une application requiert une grande puissance de calcul et travaille avec un volume de données important, l'usage du parallélisme s'impose. Cependant, même avec un haut degré de parallélisation, il arrive que les temps de calcul restent énormes. De plus, certaines méthodes de calcul parallèle à grain fin ou moyen voient leur degré de parallélisation limité par l'accroissement simultané du volume des communications *(voir la section 2.2.1)*, et le nombre optimal de nœuds pour une application donnée peut être bien en dessous du nombre de processeurs utilisables sur la machine parallèle concernée.

Lorsque tout le parallélisme « classique » a été exploité au mieux, l'utilisation optimale des ressources disponibles ne pourra être réalisée qu'en faisant appel à des méthodes nouvelles, telles les méthodes hybrides pour les applications numériques, dans lesquelles des modules à grain fin ou moyen interagissent par des couplages lâches. Le recours au parallélisme hétérogène est alors possible, et permet d'aller au delà du degré de parallélisme offert par une seule machine parallèle, au prix toutefois d'un accroissement du coût des communications entre les modules, ce qui implique un parallélisme à gros grain à ce niveau.

Par des méthodes appropriées, il est ainsi possible de tirer parti, du parallélisme disponible, hétérogène ou non, et d'accélérer de façon notable les applications concernées. Le but de ce chapitre n'est pas de développer ces méthodes (qui par exemple, pour les applications numériques, sont du domaine de la recherche en analyse numérique), mais de mettre en évidence comment élaborer la construction de ces applications hétérogènes parallèles avec efficacité, en tenant compte à la fois des questions posées et des problèmes soulevés lors de l'élaboration des versions de la méthode hybride décrites aux *chapitres 4 et 5.*

Après l'exposé des choix envisageables pour cette méthode hybride, aussi bien pour son adaptation à d'autres configurations parallèles que pour un choix pragmatique des nombreux paramètres qui conditionnent ses performances, ce chapitre exposera le problème général des applications parallèles hétérogènes. La modularité nécessaire à la construction

de ces applications sera mise en évidence en s'appuyant sur l'exemple de la méthode hybride qui, possédant des parties séquentielles et des parties parallélisables de différentes manières, en est l'exemple type. Une modélisation quant aux qualités requises des composants logiciels, aux langages de programmation, aux types de parallélisme et à l'expression de l'asynchronisme sera ensuite esquissée.

6.1 Préalables à une implantation parallèle hétérogène

La construction de la version parallèle hétérogène asynchrone exposée au *chapitre 4* et celle de la version parallèle entrelacée exposée au *chapitre 5* ont été établies en tenant fortement compte des architectures et des performances relatives des machines présentes sur les réseaux d'implantation. Cette façon de procéder est pragmatique dans le cadre d'une utilisation unique de l'application sur un réseau donné, mais handicape grandement sa portabilité vers d'autres sites puisqu'il faudrait à chaque fois construire une version adaptée. Or cette portabilité est primordiale, les machines vieillissant vite et évoluant constamment.

Afin d'être capable d'élaborer diverses versions articulées différemment, selon les matériels disponibles et les choix du programmeur pour tenter de tirer parti au mieux du parallélisme et de l'hétérogénéité, sans reprendre l'étude à zéro à chaque nouvelle version, il est nécessaire de :

- découper l'application en « briques » élémentaires, chacune d'elle étant un composant logiciel de base, séquentiel ou parallèle, pouvant être décliné en autant de versions qu'il existe d'architectures de machines capables de l'accueillir d'une manière efficace ;

- établir l'algorithme parallèle hétérogène théorique, c'est à dire la méthode de parallélisation la plus adaptée aux différentes phases de calcul de l'application, sans tenir compte des machines existantes.

Ces deux opérations étant réalisées, élaborer une nouvelle implantation de l'application consistera à choisir pour chaque module de calcul la version répondant au modèle d'architecture disponible le plus proche du modèle théorique, compte tenu d'arbitrages liés aux performances respectives des machines concernées ou pouvant prendre en compte d'autres critères d'optimisation.

La *figure 5.2*, représentant le schéma de principe de la version parallèle entrelacée de la méthode hybride, est très éloignée de la représentation de l'algorithme théorique de cette méthode car elle montre en fait le choix d'implantation qu'est l'entrelacement. Par contre, la *figure 4.1*, représentant le schéma théorique de la méthode hybride, en est très proche. Cependant, la partie séquentielle de la méthode GMRES n'y apparaît pas en raison de l'implantation SPMD de cette méthode sur une machine aux processeurs rapides qui avait

rendu préférable une redondance de la partie séquentielle pour limiter les communications.

Le détail de la construction modulaire de la méthode hybride sera abordé dans la *section 6.5.1 (voir en particulier la figure 6.1)*. À cette occasion, le modèle théorique convenant le mieux à chaque module logiciel sera précisé. Cette construction modulaire permettrait de tester d'autres types d'implantation que ceux qui ont été expérimentés. L'intérêt et les spécifications de ces autres versions sont développés ci-dessous.

6.2 Autres versions possibles de la méthode hybride

En plus des solutions hybrides parallèles déjà développées, c'est à dire l'implantation hétérogène asynchrone et l'implantation entrelacée, d'autres possibilités intéressantes sont envisageables. Selon les ressources disponibles, notamment en termes de machines parallèles, il est possible de trouver une ou plusieurs solutions adaptées. La liste exposée ci-après des solutions alternatives à celles déjà étudiées n'est pas exhaustive.

Solution asynchrone hétérogène partitionnée

La machine chargée d'exécuter la méthode GMRES peut ne pas être utilisée au maximum de ses capacités. En effet, pour des matrices de taille moyenne, les performances ne sont pas forcément les meilleures quand tous les processeurs sont utilisés, c'est à dire quand la granularité du parallélisme est maximale *(voir la rubrique parallélisation de la section 2.2.1, et la figure 2.11)*. On peut alors profiter de la partie non utilisée de la machine hébergeant GMRES en affectant les processeurs libres au calcul des valeurs propres par la méthode d'Arnoldi.

Naturellement, il faut que cette machine parallèle soit partitionnable et puisse dérouler simultanément les deux algorithmes. Ce n'est évidemment pas le cas d'une machine SIMD, et pour une machine gérée pour ne dérouler que des processus SPMD, il faudrait avoir recours à un artifice consistant à inclure les deux algorithmes dans le programme à exécuter sur chaque processeur, et à sélectionner la partie à effectuer réellement à l'aide d'un test fonction du numéro du processeur.

Avec les ressources du LIFL utilisées pour implanter la version asynchrone hétérogène décrite au *chapitre 4*, on peut très bien affecter les stations de la ferme d'Alphas non utilisées par la méthode GMRES au calcul des valeurs propres en remplacement de la MasPar. Les stations Alphas étant toutes indépendantes, il est très facile d'exécuter concurremment ces deux algorithmes sur des stations différentes.

Solution asynchrone partitionnée à redondance

La solution précédente, implantée sur une machine MIMD, peut être simplifiée lorsque les processeurs élémentaires de la machine parallèle sont suffisamment performants. Il est

alors possible d'éviter de déporter les calculs séquentiels (factorisations QR pour GMRES et pour Arnoldi) en les exécutant sur chaque PE de façon redondante. Cette possibilité évite l'utilisation de deux machines séquentielles et les transferts de données au retour du calcul. Cependant des échanges de données entre les PEs sont toujours nécessaires pour le lancement du calcul, afin que chaque PE ait connaissance de la globalité de la matrice H *(voir les sections 2.2.1 et 2.2.2).*

Naturellement, une machine séquentielle est toujours nécessaire pour l'élaboration des paramètres « Least Squares » afin que ceux-ci puissent être calculés concurremment au déroulement des autres parties de la méthode hybride (la réception de ces paramètres par le processus GMRES/LS n'est pas bloquante).

Cette solution pourrait s'avérer particulièrement intéressante pour certaines machines parallèles dont chaque nœud de calcul dispose à la fois d'un processeur séquentiel et d'un processeur vectoriel, les parties traitées ici de façon redondante pouvant facilement donner lieu à une implantation vectorielle.

Solution entrelacée à tronc commun

Une option de la version entrelacée consisterait à effectuer une seule fois par itération une partie qui est exécutée par les deux algorithmes parallèles : la « projection d'Arnoldi » *(voir les algorithmes 2 page 28, et 3 page 31).* Cette partie est très coûteuse car la complexité des calculs y est en $\mathcal{O}(nm^2 + ncm)$ [9] *(les paramètres m, n et c sont décrits par la figure 4.4).* Il serait donc intéressant de ne pas la dupliquer, mais cette économie de calcul serait sanctionnée par plusieurs contraintes supplémentaires :

- la taille m des sous-espaces de Krylov devrait être la même pour GMRES et pour Arnoldi, puisque la projection ne serait effectuée qu'une fois. Cela impliquerait une souplesse moins grande pour le paramétrage. En effet, l'ajustement des paramètres m(GMRES) et m(Arnoldi) permettait, dans la méthode entrelacée, de réduire les pertes de temps dues aux synchronisations *(voir la section 5.3).*

- la structure du programme exécuté par la machine parallèle serait plus complexe que dans la version décrite au *chapitre 5.* En effet, ce programme comporterait une partie non entrelacée, commune aux algorithmes GMRES(m) et Arnoldi, et l'entrelacement du reste des parties parallèles serait moins efficace du fait que les appels aux parties séquentielles se feraient alors au même moment (juste après la fin de la projection d'Arnoldi) pour les deux algorithmes. Le recouvrement des attentes resterait cependant partiellement possible du fait que ces deux calculs séquentiels n'ont pas la même durée, car le calcul des valeurs propres est beaucoup plus long.

- le vecteur initial serait le même pour GMRES et pour Arnoldi ce qui modifie un peu la méthode sur le plan de l'analyse numérique, en particulier pour le calcul des valeurs propres.

Cette version présenterait donc une économie de calculs importante, mais des synchronisations plus pénalisantes car la possibilité de recouvrir les attentes par des calculs serait réduite. Étant donné la lourdeur des calculs économisés, il est probable que son bilan serait globalement positif. Il serait intéressant de la mettre en œuvre pour s'en assurer.

6.3 Choix des paramètres de la méthode hybride

Comme cela a déjà été décrit aux *chapitres 4 et 5*, de très nombreux paramètres conditionnent le comportement de la méthode hybride et il n'est pas simple d'en choisir les valeurs. Penser être capable de déterminer avec finesse la valeur optimale de chacun d'eux est utopique, mais il est possible d'opérer des choix pragmatiques permettant un déroulement satisfaisant de l'algorithme avec une convergence rapide. Or, c'est justement ce dont a besoin l'utilisateur de la méthode qui doit avoir à sa disposition des critères de décision pratiques pour une mise en œuvre facile. Le but de cette heuristique est donc de permettre un paramétrage rapide et efficace, mais pas forcément optimal.

Les paramètres exposés par la *figure 4.4* (hormis n et c qui sont les caractéristiques de la matrice creuse décrivant le système linéaire à résoudre) peuvent être choisis en tenant compte des considérations suivantes :

m**(GMRES)** est choisi de la même façon que pour la méthode GMRES(m) pure, selon l'expertise de l'utilisateur;

m**(Arnoldi)** doit être choisi en tenant compte des remarques sur les tailles des espaces de projection énoncées dans les *sections 4.3.3 et 5.4.3*. En particulier ce paramètre influe sur la vitesse relative du module de la méthode d'Arnoldi par rapport aux autres (de ce fait son ajustement est plus délicat dans la version entrelacée). Une valeur un peu plus grande que celle de m(GMRES) est une bonne base de départ;

k **et** l ont une influence observable sur la *figure 4.13* et décrite à la *section 4.4.3*. Dans les tests effectués, des choix tels que $k = 15$ et $l = 5$ ont donné en général des performances satisfaisantes;

p détermine le degré de parallélisme de la partie GMRES(m)/LS et sa valeur optimale est forcément voisine de celle de la méthode GMRES(m) pure qui en constitue la partie la plus coûteuse aussi bien pour les communications que pour les calculs *(la figure 2.11 montre l'influence de p sur la méthode GMRES(m))*. Les valeurs de contrôle de la distribution des données, NB_LINES et NB_LAST, se calculent aisément à partir de n et p (valeurs utilisées : NB_LINES $= \frac{n}{p} + 1$ et NB_LAST $= n - (p-1) \times$ NB_LINES, avec un découpage de la matrice par bandes, mais d'autres types de distribution des données sont envisageables).
Si l'algorithme d'Arnoldi pour le calcul des valeurs propres est aussi implanté en mode SPMD (dans la version testée, cette partie était parallélisée en SIMD), un paramètre

$p_{Arnoldi}$, indiquant le nombre de nœuds de calculs dédiés à la partie parallèle de la méthode d'Arnoldi, sera aussi à déterminer. Étant donné l'importance de la partie commune de cet algorithme avec le précédent (la projection d'Arnoldi est présente dans la méthode GMRES(m) et dans la méthode d'Arnoldi), le choix $p_{Arnoldi} = p$ semble judicieux;

ε_g est la précision requise pour la convergence de la solution du système et sa valeur dépend des exigences de l'utilisateur, mais aussi des capacités de la machine hébergeant GMRES(m)/LS en ce domaine. La durée totale du calcul dépendra évidemment beaucoup de ce paramètre. Par défaut et pour une précision optimale des solutions, on peut affecter à ε_g la valeur de l'epsilon machine;

ε_a est certainement plus délicat à choisir *(voir le paragraphe sur la précision du calcul des valeurs propres de la section 4.3.3)*. Il est en effet très difficile de savoir à l'avance, pour une matrice donnée, s'il sera possible d'obtenir suffisamment de valeurs propres en un temps raisonnable pour une valeur fixée de ε_a. Mais une valeur assez grande de ce paramètre (par exemple $\varepsilon_a = 1$), malgré le manque évident de précision qu'elle induit, donne souvent cependant un apport suffisant de la méthode hybride, par rapport à la méthode GMRES(m) pure, pour accélérer la convergence de manière significative.
Une modification dynamique de ce paramètre est envisageable, afin d'obtenir un comportement adaptatif du programme en fonction des valeurs propres trouvées ou non lors des premières itérations;

ITERMIN permet de laisser la méthode GMRES(m) « digérer » les pics induits par les itérations « Least Squares » *(voir page 68)*. Le choix ITERMIN = 3 a semblé la plupart du temps suffisant lors des tests effectués;

RMIN est le nombre minimum de valeurs propres à trouver avant de changer de vecteur initial pour la méthode d'Arnoldi. Ce nombre doit être assez faible pour autoriser un changement fréquent afin d'obtenir une bonne répartition des valeurs propres trouvées. Sa valeur n'est pas critique (par exemple, RMIN = 5);

NVPMIN est le nombre minimum de valeurs propres à trouver avant chaque prise en compte. Ce nombre doit être suffisant pour que les itérations « Least Squares » soient efficaces. Mais une valeur trop grande espacerait exagérément les couplages entre les différents modules, et rendrait l'hybridation des méthodes numériques inopérante. NVPMIN = $\frac{m(\text{Arnoldi})}{2}$ paraît un bon compromis.

6.4 Problèmes posés

La grande diversité des configurations parallèles existantes et la nécessité d'y adapter les applications d'envergure, en particulier quand elles sont hétérogènes, induit l'émergence de multiples versions d'une même application telles que celles exposées dans la *section 6.2*

et dans les *chapitres 4 et 5*. Pour concevoir de telles applications, il convient de se poser les bonnes questions, en particulier afin d'en structurer les composants pour faciliter leur portage dans d'autres environnements parallèles.

6.4.1 Type de parallélisme

L'approche pragmatique d'un problème, telle qu'elle se conçoit dans un centre de calcul, conduit à se poser en premier lieu, pour une application donnée, la question : quel parallélisme adopter ? Ou plus exactement :

- quels types de parallélisme sont disponibles sur le réseau ?
- parmi ceux-ci, lesquels conviennent le mieux à mon application ?

Dans une architecture SIMD, chaque processeur élémentaire est associé à une mémoire locale, et les données sont réparties dans les mémoires de ces différents processeurs. Tous ceux-ci exécutent ensemble la même instruction, mais sur des données différentes. D'autre part, un concept de localité important régit les communications entre processeurs élémentaires : Une communication entre deux de ces processeurs voisins est très rapide, alors qu'une communication entre deux processeurs élémentaires distants est coûteuse.

Un algorithme utilisable en SIMD doit donc :

- utiliser une structure de données régulière (type matrice ou vecteur par exemple) pouvant être répartie sur l'ensemble des processeurs élémentaires.
- exécuter de nombreuses opérations concernant l'ensemble (ou un sous-ensemble important) des données de cette structure.
- gérer la répartition des données et les communications de telle façon que l'essentiel des communications soit entre processeurs élémentaires voisins.

Dans une architecture MIMD, l'application est découpée en portions de code séquentiel, avec les données correspondantes, réparties entre les différents processeurs, chacun ayant une mémoire locale et exécutant sa propre portion de code. Toutes les communications entre processeurs se font par envois de messages. Ces communications sont donc très coûteuses.

L'intérêt par rapport aux solutions précédentes est la disparition de l'obligation d'exécuter des traitements identiques sur des données bien calibrées. Mais cette plus grande souplesse se paye par un coût de communications qui peut être exorbitant, surtout si, comme cela est fréquent, le débit du réseau est faible par rapport à la puissance des processeurs (c'est le cas notamment pour les stations de travail séquentielles associées en réseau). Pour qu'un algorithme MIMD soit viable, il devra donc réduire au minimum les communications. Cependant, si on veut profiter des ressources disponibles pour exécuter

des tâches distinctes, ce type de parallélisme reste la seule solution envisageable.

Cette possibilité n'est pas toujours utilisée d'ailleurs, car beaucoup de programmes fonctionnant sur ces machines relèvent du modèle SPMD. Il s'agit d'un cas particulier important, pour lequel seules les données sont réparties, le même code étant dupliqué sur tous les processeurs. Il permet d'exprimer le parallélisme de données sur des machines dont l'architecture est plutôt associée au parallélisme de tâches.

Il faut reconnaître que les constructeurs s'orientent de plus en plus vers des machines du type MIMD, ce type de parallélisme pouvant même s'obtenir à un prix modique en mettant en réseau des stations de travail séquentielles. Les performances sont souvent modestes, non seulement en raison du débit insuffisant du réseau, mais aussi du goulot d'étranglement causées par des entrées-sorties séquentialisées et non optimisées. Cependant l'argument économique joue en leur faveur, et au rythme actuel de l'accroissement des performances à la fois des stations de travail et des réseaux, il n'y a aucun doute que cette solution ne prenne de l'ampleur.

En ce qui concerne les applications numériques, il s'agit très souvent de traitements lourds opérant sur des structures de données de grande taille fortement communiquantes. Le parallélisme de données est alors en général bien adapté, sauf lorsque les dépendances rendent le bénéfice de la parallélisation incertain. Ce parallélisme de données pourra être mis en œuvre soit sur une architecture SIMD, soit par un parallélisme de tâches SPMD (mais au prix d'une plus grande lourdeur dans les communications qui induira une granularité plus grosse). Certaines parties d'applications notamment numériques peuvent aussi avec avantage être traitées par des calculateurs vectoriels. Ce cas n'a été testé, aucune machine vectorielle n'étant présente sur notre réseau.

Dans le cas général, une application hétérogène comprendra plusieurs modules, fonctionnant sur le principe du parallélisme de tâches, chacun de ces modules pouvant être soit séquentiel, soit parallèle voire vectoriel selon le modèle le plus approprié à son contenu.

6.4.2 Communications

Dans un réseau hétérogène, les communications sont à plusieurs niveaux :

- les communications entre machines, en général acheminées par un réseau peu performant, et dont il est préférable de limiter le volume.
- les communications internes à une machine parallèle, souvent acheminées par un réseau rapide.
- de plus certaines machines présentent des performances différentes pour les communications entre processeurs voisins et entre processeurs distants, les communications

de voisinage étant plus rapides.

Ces différents types de communications doivent absolument être pris en compte dans la répartition des processus d'une application hétérogène, sous peine de rendre la parallélisation complètement inefficace.

De plus, le délai d'acheminement des messages doit parfois être considéré : si le message α part avant le message β d'un processeur x vers un processeur y, il n'est pas du tout certain qu'ils arrivent dans le même ordre, et y pourrait recevoir β avant α. Naturellement, ce désordre peut intervenir à plus forte raison lorsque les messages proviennent de processeurs distincts. Lorsque l'ordre de réception a une importance, un estampillage devra être effectué, afin que le processus récepteur puisse reclasser les messages reçus.

6.4.3 Synchronisation

Une particularité du parallélisme de tâches est l'asynchronisme des processus engagés. Cet asynchronisme peut être un avantage car il permet de gagner du temps, en éliminant les attentes. Cependant l'existence de dépendances impose l'ajout de points de synchronisations, notamment quand une tâche attend des paramètres calculés par une autre. Des synchronisations plus élaborées sont même parfois nécessaires quand plusieurs processus communicants doivent impérativement passer par la même phase de calcul en même temps *(voir la section 4.3.2 et l'algorithme 7 de la figure 7.4)*.

Certaines de ces synchronisations peuvent être sources de blocages (« dead-locks ») lorsque plusieurs processus s'attendent réciproquement, ou si un processus en attend un autre qui n'entrera jamais dans la phase de synchronisation.

La définition précise des besoins de synchronisation d'une application parallèle, hétérogène ou non, est donc une étape primordiale préalable à son implantation. En particulier, les possibilités de blocage inter-processus doivent être détectées et les synchronisations aménagées pour éviter tout risque potentiel.

6.4.4 Charge des machines

Beaucoup de sites hébergeant des machines parallèles fonctionnent en temps partagé, et les performances d'une application parallèle implantée sur ces sites varie énormément en fonction du nombre d'utilisateurs simultanés des machines concernées. Le caractère aléatoire de la charge des machines rend très difficile l'évaluation des performances d'une application fonctionnant dans ces conditions. Non seulement la charge évolue d'un essai à l'autre, mais elle évolue constamment, et souvent dans une proportion importante, au cours d'une même période de calcul.

Cette contrainte est encore plus à prendre en considération dans le cas d'une application hétérogène, non seulement en raison du plus grand nombre de machines engagées, mais aussi parce qu'il est difficile d'envisager de mobiliser plusieurs machines séquentielles et parallèles pendant un temps quelquefois assez long (il s'agit d'applications lourdes) pour un seul utilisateur comme cela se pratique avec certaines machines parallèles dédiées.

Pour les parties fortement synchrones de l'application, les variations de charge sont très pénalisantes car la surcharge d'un seul nœud va ralentir tous ceux qui lui sont synchronisés. D'autre part les transferts de données entre deux processus asynchrones vont induire un non-déterminisme de la prise en compte de ces données lorsque la charge varie. Ce comportement du programme, perturbant pour l'esprit, n'est cependant pas pénalisant pour l'application qui peut ainsi tirer parti au mieux des données dès qu'elles sont disponible, bien que l'instant optimal du couplage entre deux modules asynchrones soit malaisé à déterminer.

De nombreuses équipes de recherche travaillent sur les problèmes de la régulation de charge et de la migration, notamment au niveau du système d'exploitation ou de surcouches de ce système [15, 28]. Les applications hétérogènes exploitant ces possibilités pourraient être beaucoup moins affectées par les variations de charge et bénéficier ainsi à la fois d'une amélioration notable de leurs performances et d'une meilleure fiabilité de la mesure de celle-ci. C'est dans cette optique que les spécifications d'un langage de contrôle basé sur le système MARS [27] sont développées au *chapitre 7*.

6.5 Modularité

Toute application parallèle hétérogène comprenant des parties séquentielles ainsi que des parties parallélisables de différentes manières se trouve de fait partagée en différents modules, chacun d'eux étant un composant logiciel adapté à un type d'architecture. En formulant convenablement cette modularité, le programmeur peut en tirer parti pour faciliter la portabilité de l'application dans d'autres environnements parallèles. C'est particulièrement vrai pour la méthode numérique hybride étudiée dans les chapitres précédents.

6.5.1 Composants logiciels de la méthode hybride

Dans les différentes solutions envisagées, en particulier dans celles déjà mises en œuvre dans les deux méthodes décrites aux *chapitres 4 et 5*, on rencontre de nombreuses parties communes dans les algorithmes employés. On retrouve, comme composants logiciels communs :

1. Composants parallèles :
   (a) la projection d'Arnoldi (pour les deux méthodes, GMRES et Arnoldi),
   (b) le calcul du résidu et la stratégie de redémarrage pour la méthode d'Arnoldi,

(c) le calcul de la solution et de sa norme résiduelle pour la méthode GMRES,

(d) le calcul de l'itéré « Least Squares ».

2. Composants séquentiels :

(a) la factorisation QR pour GMRES,

(b) le calcul des valeurs et des vecteurs propres (QR) pour la méthode d'Arnoldi,

(c) le calcul des paramètres « Least Squares ».

Il est important de ne pas avoir à tout réécrire en passant d'un type d'implantation à un autre, et de jouer la carte de la réutilisation. C'est d'ailleurs ce qui a été fait dès la conception de la première mise en œuvre de la méthode hybride en réutilisant des modules issus d'implantations précédentes de la méthode GMRES pure (composants n^{os} 1a et 1c de la liste précédente), de la méthode d'Arnoldi pure (composants n^{os} 1a et 1b), ou de bibliothèques d'algorithmes numériques (composants n^{os} 2a et 2b).

Cependant, chaque module doit être adapté à l'environnement dans lequel il va s'exécuter : une version à parallélisme de données ne peut pas être écrite comme une version à parallélisme de tâches. De plus, la gestion des communications est aussi liée au type de machine parallèle employé et aux primitives de communications disponibles sur le réseau.

Afin d'obtenir une modularité et une adaptabilité maximale de la méthode, on pourra la partager en différents composants assurant les fonctions précédemment citées. Chaque module devra être décliné en plusieurs versions selon les différents modes de parallélismes et les protocoles de communications en vigueur, afin de pouvoir assembler facilement les éléments nécessaire à une version donnée. Cet assemblage pourrait d'ailleurs se faire de façon automatique, en fonction d'un fichier de configuration contenant la déclaration des ressources disponibles (type de parallélisme, machines utilisées, compilateurs disponibles, couches de communications existantes).

Ainsi les modules parallèles (n^{os} 1a, 1b, 1c et 1d) pourront être écrits *(voir la figure 6.1)* :

- dans un langage à parallélisme de données, tel que Fortran 90 ou HPF, seules les directives de placement des données étant à reprendre en fonction de l'implantation désirée.

- dans un langage séquentiel (tel que Fortran 77) décrivant chaque processus d'une parallélisation SPMD, le nombre de processus de ce type et donc la taille des parties de tableaux reçues par ces processus étant paramétrable.

En faisant abstraction des machines disponibles et de leurs performances, il est évident que les calculs matriciels présents dans les méthodes numériques concernées, dans lesquelles beaucoup de calculs identiques sont exécutés sur un grand volume de données, se réfèrent plutôt au modèle du parallélisme de données. L'algorithme théorique des parties parallèles

n°	Modules	Codages			Modules de communications attachés
		SIMD	SPMD	séquentiel	
P1	Initialisation de GMRES et projection d'Arnoldi (dont le produit matrice-vecteur Av_j)	×	×		- Échanges de données entre processeurs, - Envoi de H vers S1.
S1	Factorisation QR pour GMRES		×	×(*)	- Réception de H de P1, - Envoi du résultat à P2.
P2	Calcul de la solution et du résidu (GMRES), Test de convergence.	×	×		- Réception du résultat du calcul QR de S1, - Envoi du résidu à P4.
P3	Calcul de l'itéré « Least Squares »	×	×		- Réception des paramètres « Least Squares » de S3.
P4	Initialisation de la méthode d'Arnoldi et projection d'Arnoldi (dont le produit matrice-vecteur Av_j)	×	×		- Réception du résidu de P2, - Échanges de données entre processeurs, - Envoi de H vers S2.
S2	Calcul QR pour Arnoldi		×	×(*)	- Réception de H de P4, - Envoi des valeurs et des vecteurs propres vers P5.
P5	Calcul du résidu et redémarrage pour Arnoldi	×	×		- Réception des valeurs et des vecteurs propres de S2, - Envoi des valeurs propres à S3.
S3	Calcul des paramètres « Least Squares »		×	×	- Réception des valeurs propres de P5, - Envoi des paramètres « Least Squares » à P3.

(*) Ces modules peuvent éventuellement être vectoriels.

FIG. 6.1 - *Codages à réaliser pour les différents modules de la méthode hybride afin d'être à même de réaliser facilement toute nouvelle implantation*

correspondra donc à la version utilisant un langage à parallélisme de données. Il est à noter que c'est aussi cette version qui permet la meilleure portabilité, ce type de langages comportant des abstractions prenant en charge le parallélisme et permettant l'adaptation à un matériel donné par un petit nombre de directives claires. Il n'est pas sûr cependant que ce soit la plus efficace, le haut niveau d'abstraction de ces langages rendant quelquefois plus difficile certaines optimisations liées au matériel.

On pourrait penser que les modules séquentiels (n^{os} 2a, 2b et 2c), sont indépendants de la version choisie et peuvent être établis une fois pour toutes. En réalité, il faut aussi prévoir le cas où il est plus intéressant de faire les calculs concernés sur chaque processeur de manière redondante, plutôt que de déporter les parties séquentielles au prix de commu-

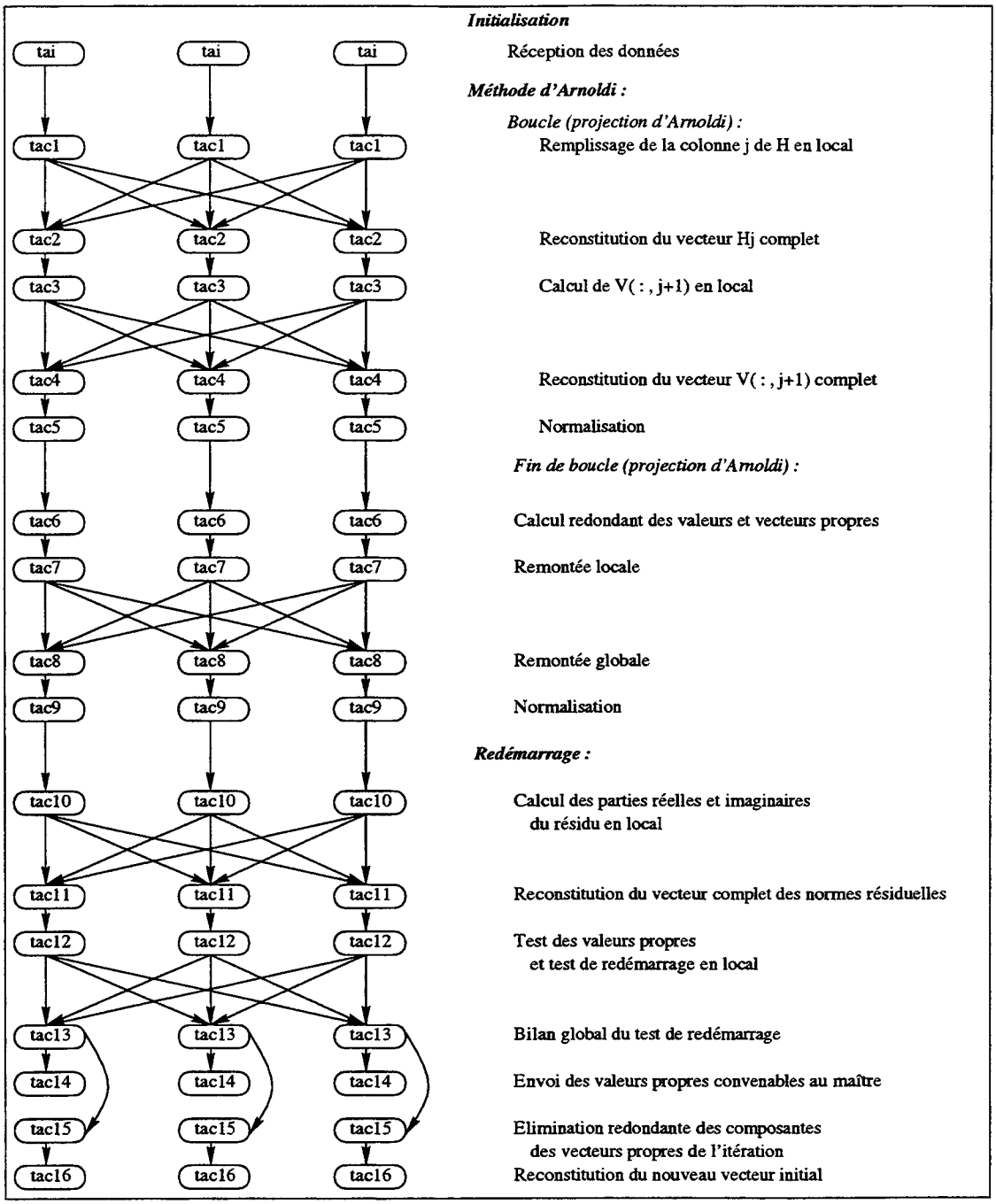


FIG. 6.2 - *Graphe de dépendances de la méthode d'Arnoldi*

nications coûteuses. Cela revient en fait à paramétrer ces processus pour qu'ils puissent fonctionner soit seuls, soit en SPMD avec une partie seulement des données. Mais c'est tout de même leur aspect séquentiel qui doit être mis en avant dans le modèle théorique du projet.

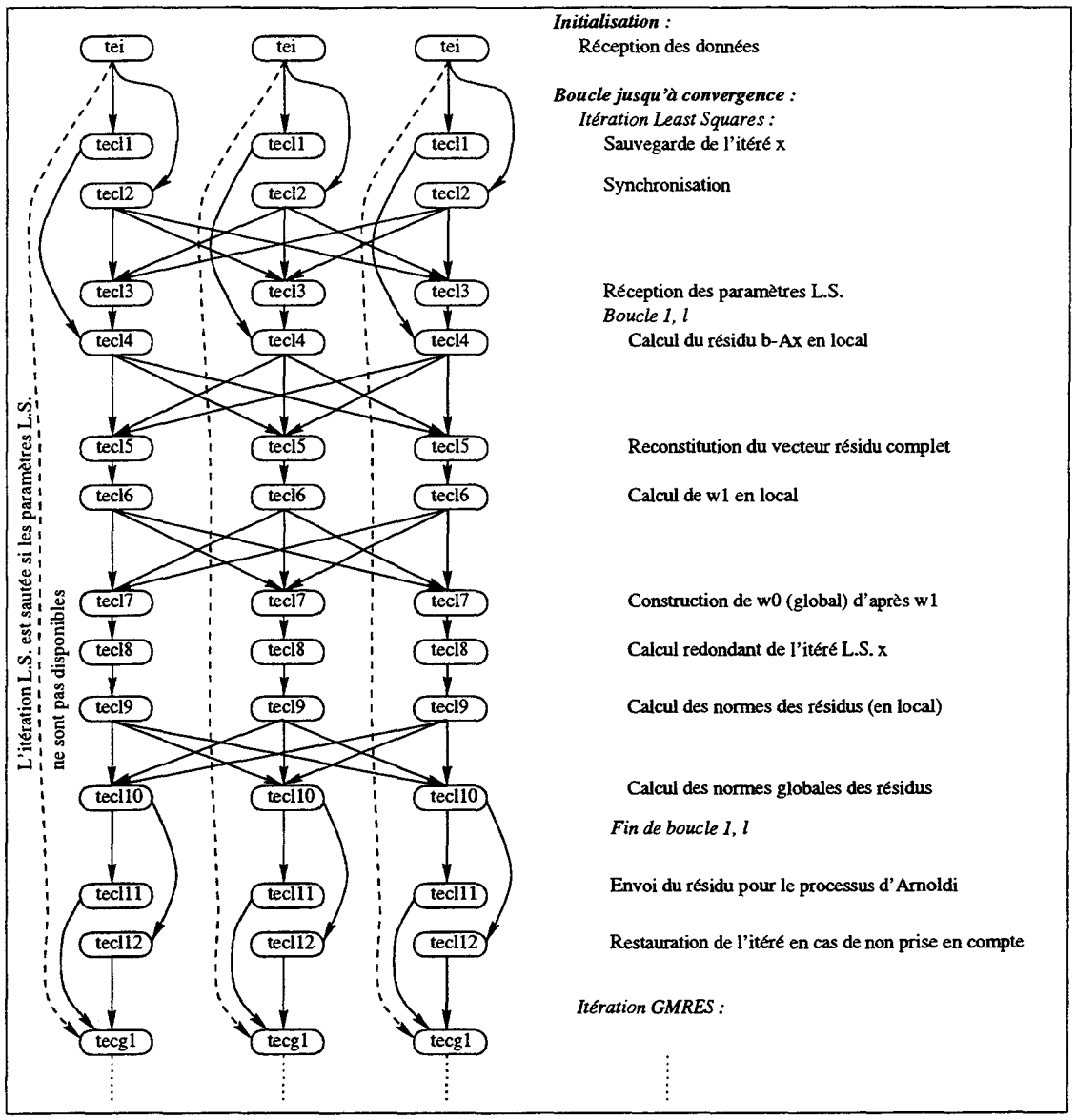


FIG. 6.3 - *Graphe de dépendances de la méthode GMRES(m)/LS(k,l) : partie 1/2 (initialisation, itération Least Squares)*

Il pourrait aussi être intéressant d'écrire ceux de ces modules qui s'y prêtent dans une version utilisable par une machine vectorielle. Cet aspect n'a pas été développé ici, aucune machine de ce type n'étant présente sur notre réseau.

Les modules servant aux communications (initialisations et retour des parties séquentielles ou initialisations des parties redondantes, échanges entre processus SPMD) dépendent évidemment étroitement du gestionnaire disponible sur le réseau. Pour les implantations décrites aux *chapitres 4 et 5*, seul PVM a été utilisé. Mais l'utilisation de

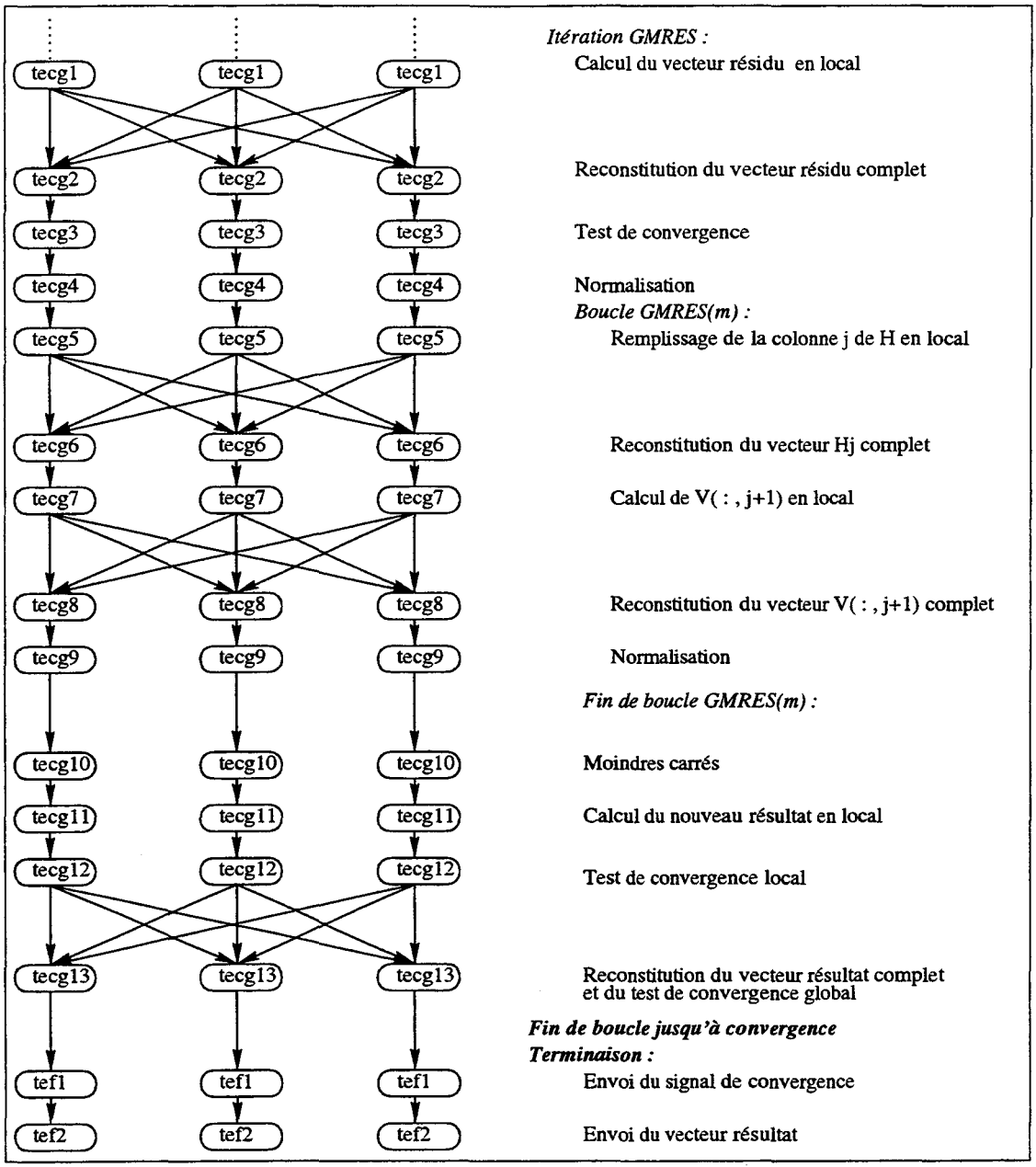


FIG. 6.4 - *Graphe de dépendances de la méthode GMRES(m)/LS(k,l) : partie 2/2 (itération GMRES, terminaison)*

bibliothèques de communications plus élaborées peut simplifier cette partie.

6.5.2 Granularité de la méthode hybride

Comme cela a été vu dans la *section 2.2.1*, le degré de parallélisme de méthodes telles que GMRES reste limité par le grand nombre des communications engendrées. La gra-

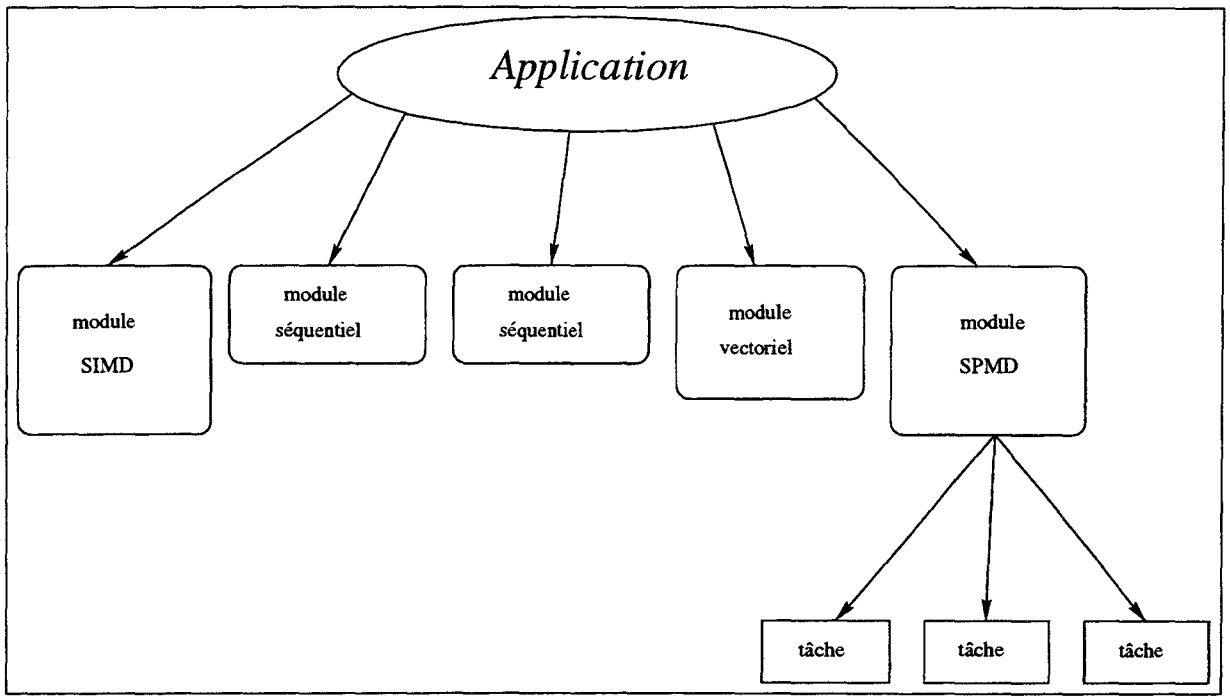


FIG. 6.5 - *Répartition des composants logiciels d'une application parallèle hétérogène*

nularité est donc assez importante, ce qui rend l'application assez sensible à la surcharge d'un des nœuds engagés, d'autant plus que de nombreuses synchronisations vont aligner la vitesse d'exécution de l'application à celle du nœud le plus chargé.

Le graphe de précédence *(voir les figures 6.2, 6.3 et 6.4)* des parties parallèles de la méthode hybride asynchrone montre une totale indépendance des processus traitant chaque sous-matrice, sauf aux moments des recontructions des vecteurs globaux échangés qui sont aussi les points de synchronisation de l'algorithme. Une telle indépendance devrait rendre plus facile la migration éventuelle de la partie de l'application affectée à un nœud chargé. Il serait possible aussi d'avoir un grain qui ne soit pas constant, et dépende de la charge courante de chaque processeur. La gestion d'un tel système est cependant complexe, et justifie l'utilisation d'un langage de contrôle adapté.

6.5.3 Qualités requises des composants

À l'instar de ce qui vient d'être vu à propos de la méthode hybride étudiée précédemment, une application hétérogène va comprendre plusieurs composants logiciels, chacun d'eux étant exécuté soit par une machine séquentielle, soit par une machine parallèle *(voir la figure 6.5)*. Dans ce dernier cas, le module, vu comme une entité unique sur le plan hétérogène, peut se subdiviser au niveau de la machine parallèle, dans le cas du parallélisme de tâches, en plusieurs composants logiciels élémentaires homogènes.

Les composants hétérogènes, représentant une première parallélisation à gros grain de l'application, doivent être tels que:

- chaque module soit bien typé quand à sa séquentialité ou au type de parallélisation qu'il supporte,
- le volume et le nombre de communications entre modules soient réduits.

En général, chaque module effectuera une partie précise de l'application, correspondant à une subdivision naturelle du calcul, et ayant une grande autonomie relativement aux autres, ce qui autorisera un couplage faible.

Contrairement à la propriété requise généralement pour la parallélisation des programmes postulant que l'application parallélisée doit faire la même chose qu'en séquentiel [21], il est possible de tirer parti du déroulement simultané des modules et de la possibilité de les coupler de manière asynchrone, pour obtenir de meilleures performances de calcul. Naturellement, l'application doit toujours donner les résultats qu'on attend d'elle (s'il s'agit de résoudre un système linéaire, par exemple, les solutions trouvées doivent être exactes à la précision demandée près). Mais le cheminement pour y parvenir peut varier de celui obtenu en séquentiel, voire peut varier d'une fois sur l'autre si le non déterminisme est toléré par l'application, c'est à dire si l'application converge vers des résultats stables quel que soit le chemin suivi dans l'ordre des calculs et le couplage des modules les exécutant.

6.6 Langages et types de parallélisme

Les langages utilisables pour l'écriture d'une application parallèle hétérogène doivent donc avoir les qualités suivantes:

- pouvoir tirer partie des différentes architectures parallèles utiles pour chaque composant logiciel du projet.
- être facilement portables, d'une part afin que le produit fini puisse être aisément inséré dans des applications existantes dont il serait l'optimisation d'un des composants, d'autre part afin de pouvoir évoluer facilement en fonction des progrès réalisés dans les architectures.

Ces deux exigences sont contradictoires. En effet, les langages classiques sont souvent trop généraux pour tirer complètement partie d'une architecture particulière. C'est pourquoi les constructeurs ont souvent développé des langages mettant en valeur les spécificités de leurs machines. Par définition ces langages ne sont hélas pas portables.

La plupart des langages permettant de traiter le parallélisme de données appartiennent à deux familles: les extensions de Fortran 77 et les extensions du C [33]. Ces langages modélisent en général les données parallèles sous la forme d'une extension du type tableau.

Dans les extensions de Fortran, il y a même absence de distinction entre les tableaux classiques et la classe des données parallèles, le choix étant fait par le compilateur en fonction des opérations requises par ces données. Cette assimilation peut être une qualité car elle permet de ne modifier que très peu le texte source du programme pour passer d'une architecture séquentielle à une architecture parallèle.

Un critère important est le placement des données, c'est à dire la manière de distribuer les données sur les différents processeurs de la machine parallèle. Ce placement est déterminant aussi bien en terme de granularité du parallélisme que de gestion des communications, et donc d'efficacité. Certains langages ont des directives de placement calquées sur la géométrie du parallélisme qu'ils gèrent (par exemple MPL pour la MasPar), ce qui les rend très efficaces, mais peu portables. D'autres utilisent un placement implicite (les extensions de Fortran), ce qui facilite leur portabilité, mais fait perdre au programmeur le contrôle du parallélisme [31]. Pour y remédier, certains compilateurs placent les directives liées à une architecture particulière dans une ligne de commentaire, ce qui permet leur portabilité, sans résoudre complètement le problème car sur une autre machine parallèle, il faudra donner d'autres directives de placement. C'est le procédé utilisé notamment par MPFortran (le Fortran de la MasPar) pour lequel une ligne commençant par CMPF signale une instruction spécifique MasPar (qui sera ignorée par un autre compilateur). C'est aussi le cas pour CMFortran (le Fortran de la CM5) dont les instructions propriétaires commencent par CMF$.

Le choix du langage est aussi lié au type des applications dans lesquelles va s'inscrire notre algorithme parallèle. Or de nombreux programmes scientifiques ont été écrits en Fortran, et la parallélisation de certains modules, dont les temps d'exécution sont critiques, ne doit pas obliger à réécrire l'ensemble de l'application. En effet, beaucoup d'applications coûteuses en temps de calcul passent la majeure partie de leur temps dans quelques modules critiques (de calculs matriciels par exemple), particulièrement gourmands en temps d'exécution. Le choix de ce langage est donc intéressant afin de permettre la réutilisabilité de la quasi-totalité du code existant, tout en accélérant l'exécution des parties critiques réécrites.

Dans le cas du parallélisme de tâches, il est aussi possible d'utiliser un langage séquentiel comme Fortran 77 pour écrire chaque processus SPMD, en gérant les communications ainsi que le lancement et la synchronisation des processus avec des bibliothèques de primitives dédiées à cet usage. C'est ce qui a été fait pour les processus tournant sur la ferme d'Alpha dans l'implantation décrite au *chapitre 4*, en utilisant Fortran 77 et PVM.

À défaut d'un langage permettant de gérer directement l'hétérogénéité, une application parallèle hétérogène utilise classiquement le même procédé : les modules élémentaires, séquentiels ou parallèles, sont écrits à l'aide d'un des langages appropriés à l'architecture où il est implanté. Et la gestion des processus et des communications se fait à l'aide de bibliothèques appropriées.

Cependant cette façon de procéder un peu artisanale exige du programmeur beaucoup de temps, et une vigilance particulière pour gérer les synchronisations en évitant les blocages et les confusions dans les messages. De plus, elle risque de conduire à des programmes sources compliqués et peu lisibles, dont la maintenance serait difficile et contraire au exigence du « génie logiciel ». L'utilisation de langages de contrôle adaptés à un environnement hétérogène et sachant gérer le parallélisme à deux niveaux (celui d'une machine parallèle et celui d'un réseau hétérogène) peut grandement faciliter les choses.

6.7 Expression de l'asynchronisme

La mise à profit de l'asynchronisme pour améliorer les performances d'une application est une option intéressante, mais qui choque un peu nos esprits cartésiens qui raisonnent toujours en termes de logique séquentielle : on fait une chose après une autre, ou à la rigueur on en lance deux en même temps, mais on se synchronise un peu plus loin. Évidemment beaucoup d'applications fonctionnent avec des dépendances qui obligent un processus à attendre les données provenant d'un autre. Mais il est parfois possible de profiter des informations émises par un processus complètement indépendant et asynchrone, pour enrichir les données d'une application et la rendre plus performante à mesure que ces informations arrivent. C'est le principe même de l'implantation parallèle hétérogène asynchrone décrite au *chapitre 4* de la méthode GMRES/LS-Arnoldi pour résoudre un système linéaire.

Pour que l'asynchronisme entre deux modules, ou entre deux groupes de modules, soit total, il faut évidemment qu'il n'y ait aucune dépendance entre eux. Chaque module est indépendant et est apte à réussir seul le traitement pour lequel il a été écrit. Mais l'enrichissement apporté par les informations provenant d'un autre module peut lui permettre d'arriver plus vite au résultat.

Le fonctionnement n'est pas forcément à sens unique. Deux modules peuvent s'enrichir réciproquement. C'est le cas de l'application du *chapitre 4* dans laquelle le module GMRES/LS reçoit d'une manière asynchrone les valeurs propres calculées par le module de la méthode d'Arnoldi, alors que celui-ci reçoit aussi d'une manière asynchrone le vecteur résidu de la méthode « Least Squares » qu'il utilisera comme vecteur initial lors du prochain redémarrage de la méthode.

La conséquence directe de cet asynchronisme total est le non déterminisme du calcul. En effet, lorsque les machines ont simultanément plusieurs utilisateurs qui y accèdent en temps partagé, cela induit des variations importantes dans les performances observées en fonction des travaux lancés ou arrêtés à tout moment sur ces machines. À chaque lancement de notre programme, même avec les mêmes données, les aléas du couplage des modules, dûs notamment aux évolutions imprévisibles des charges des machines, feront que chaque module ne verra pas toujours arriver au même moment de son déroulement les informations provenant des autres modules. L'instant de la prise en compte de ces données

étant variable, celles-ci ne seront pas couplées de la même façon aux données du module récepteur (dans la cas d'une méthode itérative, le couplage n'aura pas toujours lieu lors de la même itération). L'accroissement résultant des performances variera donc d'une fois sur l'autre.

Mais aucun délai d'attente n'intervenant, le fonctionnement sera donc optimisé en fonction de la charge instantanée des machines. En effet, des modules n'ayant aucune dépendance ne sont pas ralentis par l'attente de données issues de modules externes, ce qui garantit un fonctionnement indépendant limité seulement par les performances courantes de la machine hôte.

Pour implanter les réceptions de données asynchrones, on utilisera naturellement des primitives de réception non bloquantes. Mais lorsque le module récepteur est parallèle, il faut veiller à la synchronisation de la prise en compte des données reçues par les différentes tâches du module, notamment lorsqu'il s'agit de processus SPMD, afin que la cohérence des données soit maintenue entre ces tâches. Le non respect de cette synchronisation risque d'entraîner la prise en compte des données par une partie seulement des processus concernés, ce qui conduit soit à un calcul erroné, soit à un « dead-lock ».

Ces applications fortement asynchrones engendrent donc tout de même des problèmes de synchronisation, mais ceux-ci sont spécifiques et les délais d'attente qui en découlent ne sont pas pénalisants car les communications concernées interviennent rarement.

6.8 Conclusion

La construction d'applications parallèles hétérogènes fait intervenir de nombreux choix, qui rendent sa mise en œuvre complexe. La décomposition de ces applications en parties soit séquentielles, soit relevant de différents types de parallélisme, l'existence de plusieurs niveaux de communications, les problèmes de synchronisation, de variations imprévisibles de la charge instantanée des machines, rendent délicates l'élaboration et la mise au point de telles applications.

Les langages de programmation parallèles ne sont pas suffisants pour rendre compte des spécificités et de la complexité des applications hétérogènes. Mais il existe des outils, dont plusieurs sont décrits au *chapitre 3*, qui autorisent leur écriture. Comme cela a été montré à propos des différentes versions hétérogènes de la méthode hybride GMRES(m)/LS-Arnoldi, une application hétérogène se décompose en modules élémentaires séquentiels ou parallèles, chacun étant écrit dans un langage de programmation approprié, séquentiel ou parallèle selon le cas. La coordination de l'ensemble des modules, tant sur le plan des échanges de données que sur celui du lancement et de la synchronisation des tâches, sera avantageusement assurée par un langage de contrôle dont font partie certains des outils précités.

Plutôt que de devoir décliner chaque module selon différentes versions (séquentielle, SPMD, SIMD) et de prévoir pour chacun d'eux plusieurs blocs de communication dépendant des outils disponibles (PVM, MPI, ...), il serait préférable d'étendre suffisamment la modélisation du parallélisme hétérogène afin de pouvoir le décrire finement et le prendre en compte de manière plus systématique, avec des outils permettant au programmeur de cerner le problème de manière moins artisanale. Ainsi il serait intéressant de disposer, dans un langage de contrôle, de directives permettant :

- de décrire les placements de données en fonction de critères liés à l'architecture et au nombre de processeurs disponibles (dans le style de ce qui se fait en parallélisme homogène, par exemple avec HPF [29] *(voir la section 3.2.5)*),
- de différencier les communications dans un réseau rapide au sein d'une machine de celles liées à l'hétérogénéité,
- de définir des classes de modules de calcul liés par des communications intenses et ne devant pas être dispersées dans un système hétérogène,
- de décrire les synchronisations et les estampillages à mettre en place.

Cela revient à faire une programmation à deux niveaux :

- la programmation des modules élémentaires, séquentiels ou parallèles, intégrant éventuellement des éléments de bibliothèques applicatives,
- la programmation de l'architecture hétérogène, pilotant le lancement des modules et les relations inter-modules telles que communications et synchronisations.

À terme, un pré-processeur pourrait lire un fichier de description d'une application hétérogène associé à des fichiers sources rédigés dans un langage de programmation gérant le parallélisme sur des architectures diverses (style HPF). Ce pré-processeur pourrait lancer les bons compilateurs sur les bonnes machines, séquentielles ou parallèles, en associant aux programmes sources les bonnes procédures de communications, issues des bibliothèques disponibles sur le réseau d'implantation.

Afin de remédier aux inconvénients induits par les variations de la charge des machines, les spécifications d'un langage de contrôle basé sur un système permettant le parallélisme adaptatif, sont exposées au *chapitre 7.*

Chapitre 7

Expression du parallélisme hétérogène par un langage de contrôle

7.1 But

La nécessité d'avoir des outils permettant de décrire et de réaliser aisément des applications hétérogènes a été décrite au *chapitre 3*. L'utilisation de langages de contrôle élaborés permet d'alléger le travail du programmeur tout en offrant une plus grande efficacité sur la gestion des processus et des communications. Un certain nombre de ces outils ont été décrits dans la section 3.2. La plupart d'entre eux ne permettent pas de gérer l'hétérogénéité autrement qu'à un très bas niveau. De plus, aucun ne tient compte d'un facteur essentiel qui rend très difficile l'évaluation des performances d'un programme parallèle hétérogène : la variation de la charge des machines. Or il est évident que l'optimisation de l'utilisation de ressources coûteuses impose la plupart du temps des accès en temps partagé dont le volume est par nature très variable, et imprévisible.

Le but de ce chapitre est de décrire les caractéristiques d'un langage de contrôle tenant compte de ce paramètre, et adapté à des applications parallèles hétérogènes, telles que les applications numériques hybrides. Il devra donc permettre une écriture aisée des applications travaillant sur des structures de données volumineuses et fortement communicantes, et autoriser une répartition des calculs et des données qui tienne compte des architectures et des performances des composants matériels hétérogènes engagés. Il devra aussi, savoir tirer le meilleur parti de l'asynchronisme potentiel des différents composants logiciels de l'application, tel celui mis en évidence dans l'application décrite au *chapitre 4*.

7.2 Utilisation de MARS

Pour contrôler une application répartie sur différentes machines, tout en tenant compte de la charge de ces machines, il est nécessaire à la fois de disposer en temps réel d'un état de la charge du système et d'être capable de migrer les processus tournant sur les machines chargées. Des surcouches du système UNIX telles que « GatoStar » [15], ou « MARS » [28] permettent de gérer des applications parallèles distribuées en migrant les tâches en cas de surcharge.

En particulier, le système « MARS [1] » [27, 26], développé par le LIFL, autorise de modifier dynamiquement la granularité du parallélisme. Il réalise l'ajustement adaptatif du parallélisme d'applications réparties et la régulation de charge, c'est à dire que le nombre et la localisation des tâches sont dynamiques. Il semble par conséquent pouvoir constituer une base intéressante pour construire un langage de contrôle répondant aux critères définis ci-dessus.

7.2.1 Le méta-système MARS

Il s'agit d'un environnement de programmation destiné à gérer des applications parallèles distribuées sur un réseau de machines hétérogènes. MARS est construit à partir de PM^2 [35, 36], qui est une plateforme de multithreading distribué basée sur le concept d'appels de procédure à distance (LRPC). Il permet de réaliser :

- **la virtualisation des ressources** permettant d'intégrer un parc hétérogène de stations de travail, et de machines parallèles d'architectures différentes dans une vue unique de super calculateur virtuel : le méta-système MARS,
- **l'ordonnancement adaptatif** des tâches de l'application en en modifiant dynamiquement la granularité en fonction du graphe de précédence et de la charge des machines,
- **la régulation de charge** permettant d'insérer une tâche sur un nœud sous utilisé, et de la retirer dès que l'occupation de ce nœud devient trop importante,
- **la tolérance aux pannes**, avec un protocole de reprise permettant de tenir compte des calculs déjà effectués. Une sauvegarde périodique de l'application est effectuée pour permettre cette relance [30].

MARS permet de tirer parti du parallélisme disponible, y compris sur un réseau de machines séquentielles dont la puissance de calcul est souvent sous-utilisée [27]. Il est notamment possible de profiter des périodes d'inutilisation de la console d'une station de travail pour y engager un élément de calcul d'une application parallèle (dépliage de l'application), quitte à le désengager (repliage) dès que la console sera réactivée, afin de tenir compte du

1. Multi-user Adaptive paRallel Scheduler

caractère personnel de ce type de stations. Les ressources « gratuites » ainsi dégagées, par la mise à profit du temps habituellement perdu, peuvent être très importantes.

Les applications sont basées sur le modèle maître-esclaves. Les tâches esclaves sont du type SPMD et travaillent sur un volume de données variable, dépendant de la granularité liée à la charge du nœud considéré. La tâche maître gère le lancement et l'arrêt des tâches esclaves, et adapte donc le parallélisme aux ressources disponibles.

La distribution des données est assurée par le maître, et le volume alloué à chaque esclave est adapté à la charge de sa machine-hôte. Le maître se charge aussi de la récupération des données traitées au moment du retrait d'un esclave ou à la fin de son travail.

Lorsqu'une panne est détectée, la reprise est basée sur la restauration des données et le redémarrage de l'algorithme en des points de contrôle, au passage desquels des sauvegardes ont été faites.

7.2.2 Intérêt d'un langage de contrôle utilisant MARS

Outre les avantages liés à l'utilisation de tout langage de contrôle, et discutés précédemment *(voir le chapitre 3)*, l'intérêt d'en construire un à partir de MARS est justement inhérent aux avantages du système MARS :

- La régulation de charge permet notamment de s'affranchir des gros problèmes rencontrés par les applications parallèles fonctionnant sur des réseaux de machines en temps partagé. En effet, les algorithmes parallèles SPMD fortement communicants et synchronisés peuvent être considérablement pénalisés par la surcharge d'un seul nœud de calcul. Et justement, ce type d'algorithme se modélise bien selon le schéma maître-esclaves qui est celui de MARS.

- La tolérance aux pannes est très avantageuse quand on sait que le taux de pannes est assez élevé, et qu'une seule station en panne pénalise toute l'application, paralysant les processus qui attendent après les données pendantes. Sans compter que la détection de la panne est souvent malaisée, et que pour une application gourmande en temps de calcul, on mettra souvent plusieurs minutes pour s'apercevoir qu'elle est bloquée.

- La granularité adaptative permet de toujours tirer parti au mieux du parallélisme disponible, ce qui est particulièrement intéressant pour les applications dont le parallélisme intrinsèque varie fortement au cours des étapes du calcul. Par contre, la finesse du grain devra être bornée (ce que MARS autorise), lorsque le volume des communications engendrées par une augmentation excessive du degré de parallélisme risque de pénaliser les performances *(voir la figure 2.11)*.

- Le système MARS est portable. Il s'appuie sur le système d'exploitation UNIX, et fonctionne actuellement sur les architectures Sparc/SunOs, Sparc/Solaris, Alpha/OSF-1 et PC/Linux. Il peut facilement être porté sur d'autres architectures [28].

De plus, MARS étant construit sur l'environnement multithreadé PM^2, il permet de tirer parti des avantages de celui-ci.

Il est à remarquer que la méthode utilisée par MARS pour obtenir un fonctionnement adaptatif, et consistant en un système de redistribution des données entre les nœuds de calcul, s'appuie sur ce même principe de redistribution que le modèle de parallélisme adaptatif explicité par Guy Edjlali [9] à propos d'algorithmes numériques opérant sur des matrices creuses. Ce principe semble donc bien adapté à notre problème.

7.2.3 PM^2

PM^2 est un environnement de programmation parallèle[2] basé sur le concept de processus légers (« threads ») [35, 36]. Par rapport aux processus UNIX, dits processus lourds car leur contexte (segment de données, code, pile, ...) peut être très important, les threads nécessitent peu de ressources car un grand nombre d'entre eux peuvent s'exécuter à l'intérieur d'un seul processus Unix dont ils utilisent le contexte.

PM^2 offre des fonctionnalités basées sur la notion de « service », c'est à dire l'exécution d'une procédure ou d'une fonction à distance dans un contexte distribué. Ces services sont formalisés dans le LRPC (Lightweight Remote Procedure Call), qui permet à un thread s'exécutant sur une machine donnée, d'appeler une fonction qui s'exécutera dans un thread créé sur une machine distante. Trois types de LRPC sont proposés :

- le LRPC synchrone (**LRPC**) bloque l'appelant dans l'attente du retour du résultat.
- le LRPC asynchrone (**ASYNC_LRPC**) n'attend pas de résultat en retour. Appelant et appelé s'exécutent parallèlement.
- le LRPC à attente différée (**LRP_CALL**) permet à l'appelant de poursuivre son exécution parallèlement à l'appelé, et de définir le moment où il doit recevoir les résultats (avec attente éventuelle s'ils ne sont pas encore arrivés : **LRP_WAIT**).

Le corps du LRPC constitue sa partie « service ». Il est délimité par :

```
LRPC_SERVICE(nom_du_LRPC)
        ⋮
   (instructions)
        ⋮
END_SERVICE(nom_du_LRPC)
```

Un thread sera créé à chaque appel pour la durée de l'exécution de ce service, qui pourra

2. PM^2 signifie Parallel Multithreaded Machine

accéder à toutes les ressources du contexte du processus dont il dépend.

Les paramètres d'entrée et de sortie des LRPC correspondent en fait à des transferts de données entre processus (et donc entre machines si ces processus sont distants), gérés par PVM. D'ailleurs, sous PM^2, les communications se font grâce à des LRPC. L'empaquetage et le désempaquetage PVM de ces données doivent être décrits par le programmeur dans des « souches » : Quatre souches sont nécessaires pour chaque LRPC :

- PACK_REQ_STUB contient les instructions d'empaquetage des paramètres d'entrée,
- UNPACK_REQ_STUB contient les instructions de désempaquetage des paramètres d'entrée,
- PACK_RES_STUB contient les instructions d'empaquetage des paramètres de sortie,
- UNPACK_RES_STUB contient les instructions de désempaquetage des paramètres de sortie.

PM^2 permet la migration des threads d'une machine sur une autre, entre deux processus UNIX identiques afin de retrouver le même contexte. Cette migration est peu coûteuse car seul le contexte du thread doit être transporté.

Toutes ces propriétés permettent de gérer facilement, à l'aide de PM^2, des applications nécessitant un degré de parallélisme important. Les LRPC, notamment, autorisent un parallélisme à grain fin qui serait difficilement réalisable avec les processus lourds d'UNIX. De plus, le mécanisme de migration permet de redistribuer dynamiquement une application parallèle en fonction de la charge instantanée des machines. L'ordonnancement adaptatif de MARS en a tiré le meilleur parti.

7.3 Primitives proposées

7.3.1 Remarques préliminaires

Il est à souligner que les applications hybrides tirant parti de l'asynchronisme de leurs composants, telles que celle décrite au *chapitre 4*, sont particulièrement bien adaptées pour profiter des avantages de la régulation de charge et de la granularité adaptatives de MARS. Cependant, quelques problèmes apparaissent :

- certains modules fortement communicants risquent d'être pénalisés par le principe d'un système maître-esclaves dans lequel toutes les communications entre esclaves doivent transiter par le maître.
  À cause de la philosophie adaptative mise en œuvre, seul le maître connaît où se trouve ses esclaves et quel est le volume des données traitées par chacun. Une communication directe entre esclaves, pour éviter le goulot d'étranglement constitué par

le maître lors de l'échange de gros volumes de données, n'est envisageable qu'à condition de demander au maître les informations indispensables. Encore faut-il que la répartition et la localisation des esclaves ne changent pas au cours de la communication.
Pour ces raisons, il paraît indispensable de mettre en place un protocole permettant que, sur la requête d'un des esclaves, l'application entre dans une section critique au cours de laquelle la régulation de charge et la granularité adaptatives seraient inhibées. Le maître pourrait alors fournir à ses esclaves les informations leur permettant de communiquer directement, par les méthodes les plus efficaces. Ceux-ci fourniraient ensuite au maître un signal de fin de communication, permettant à l'application de sortir de la section critique et de profiter à nouveau de l'adaptativité de MARS.
Ce protocole peut être encapsulé dans l'une des primitives du langage de contrôle, afin d'être transparent à l'utilisateur.

- dans une application hétérogène fortement asynchrone, tous les modules ne font pas la même chose. Certes, on peut toujours ramener un ensemble de modules MIMD au modèle SPMD par des tests judicieusement placés dans le code. Mais c'est alors renoncer à exploiter la spécificité des architectures engagées, pour lesquelles certaines parties de l'application peuvent être plus ou moins bien adaptées. Le langage de contrôle devrait permettre d'engager les modules faiblement dépendants et asynchrones, concernant notamment des architectures différentes, comme une association de plusieurs applications MARS communicant entre elles.

- l'obligation de passer par le langage C pour utiliser MARS et PM^2 est pénalisante, notamment pour les applications numériques qui font un grand usage de bibliothèques mathématiques la plupart du temps écrite en Fortran. Ce problème n'est pas si simple qu'il n'y paraît, il ne suffit pas ici d'intégrer quelques fonctions C dans un exécutable à l'édition des liens : l'utilisation des LRPC pour tranférer des données entre processus exige à elle seule une construction particulière du programme source. Ce point sera discuté dans la *section 7.4.1.*

7.3.2 Primitives

Les primitives nécessaires au contrôle d'applications parallèles hétérogènes, telles que celles développées pour la mise en œuvre des méthodes numériques itératives évoquées dans les chapitres précédents, peuvent être classées en plusieurs catégories :

- des primitives de gestion de modules hétérogènes, voire asynchrones,

- des primitives de gestion de processus SPMD homogènes à l'intérieur d'un de ces modules,

- des primitives de communication et de synchronisation entre processus d'un même module (via le maître ou non),

– des primitives de communication et de synchronisation entre modules hétérogènes.

Gestion de modules hétérogènes

Il s'agit ici de pouvoir enrôler plusieurs applications MARS parallèles, complétées éventuellement par quelques processus séquentiels, dans un seul ensemble coordonné constituant l'application hétérogène.

Chaque élément de cet ensemble, appelé « module » dans la suite, est donc soit un processus séquentiel unique, soit un système maître-esclaves homogène, dans tous les cas gérés par MARS. Le mot homogène signifie ici qu'un seul type de parallélisme est employé, soit par l'utilisation d'une machine massivement parallèle, soit par la répartition de processus SPMD sur une machine parallèle MIMD ou sur un réseau. Mais si les performances de ce dernier le permettent, il n'est pas exclus que les différents processus d'un même module soient répartis sur des machines séquentielles ayant des processeurs éventuellement différents.

Un super-maître peut jouer le rôle d'arbitre, lançant ou arrêtant les modules selon les besoins, et acheminant les communications inter-modules sans que chaque module ait besoin de connaître la localisation des autres. Le couplage étant lâche entre les modules, le volume de ces communications doit rester faible, et le transfert via le super-maître ne sera pas pénalisant.

Ce super-maître pourrait utiliser les primitives suivantes :

new_module : création d'un module.
Outre la désignation du module, un paramètre indispensable est le nombre maximum d'esclaves pouvant être engagés pour ce module : ce sera un dans le cas d'un module séquentiel, et la valeur fournie pour un module parallèle dépend de la granularité maximale désirée, au-delà de laquelle le volume des communications devient trop pénalisant.

exit_module : arrêt d'un module.

Il est à noter que la version actuelle de MARS ne permet pas la coexistence de plusieurs utilisateurs MARS sur la même station. Cette limitation, justifiée par le désir de contrôler plus facilement la charge des machines, n'est pas gênante dans une application même hétérogène : il suffit de veiller à ce que les maîtres de chacune des applications MARS couplées dans l'application globale soient lancés sur des machines distinctes, voire des nœuds distincts d'une même machine parallèle.

Par contre, cela pose un sérieux problème si un autre utilisateur MARS a déjà lancé sur le réseau son application, interdisant par la même l'accès de certaines stations à ma propre

application MARS (ainsi qu'à celles des autres). Si l'application déjà lancée utilise un volume de données important ou un grain fin de parallélisme, beaucoup de machines peuvent ainsi devenir momentanément inaccessibles aux utilisateurs MARS, pénalisant gravement les applications concernées. Une nouvelle version du système, levant cette limitation, est actuellement en cours de développement.

Gestion de processus homogènes

Les processus homogènes sont ceux du même module, et constituent une application MARS. Les primitives permettant la gestion de ces processus sont donc les primitives prévues à cet effet par l'environnement MARS, c'est à dire :

- du côté du maître :

  mars_spawn : création d'un certain nombre de processus esclaves SPMD, Le nombre minimal et le nombre maximal d'esclaves à créer sont fournis en paramètre, et la fonction retourne le nombre de processus réellement créés, en fonction de la disponibilité des stations hôtes.

  mars_exit : arrêt du processus.

- du côté de chaque esclave :

  mars_startworkerthread : création du thread de l'esclave.

  mars_wait_exit : mise en attente de l'esclave, jusqu'à l'appel de **mars_exit** par un autre thread.

  mars_exit : arrêt du processus UNIX dont dépend l'esclave, et donc de tous les threads en dépendant.

Communications entre processus homogènes

En plus des communications liées au dépliage et au repliage adaptatif (« folding » et « unfolding ») de l'application MARS (les données à traiter sont réparties et envoyées par le maître au moment de la création des processus esclaves), les applications fortement communicantes comme le sont certaines méthodes numériques itératives ont besoin de faire des transferts directs de données d'un processus de calcul à l'autre, en évitant le goulet d'étranglement du transit obligé par le maître.

Pour respecter les conditions de cohérence de l'application qui imposent qu'aucun changement dans la répartition des processus de calcul n'intervienne pendant une phase de communication, les primitives permettant le transfert direct des données d'un esclave à l'autre sans passer par le maître devront permettre de réaliser simplement l'*algorithme 6 (figure 7.1)*.

ALGORITHME 6 : COMMUNICATION ENTRE ESCLAVES

1. L'esclave demande une communication directe avec ses pairs,
2. Le maître mémorise cette demande, et inhibe les fonctions adaptatives de MARS, c'est à dire toute migration, pliage ou dépliage de l'application,
3. Le maître envoie à l'esclave demandeur le tableau de tids de ses pairs,
4. Les esclaves communiquent,
5. L'esclave demandeur indique qu'il a terminé,
6. Le maître efface la demande de communication. Si aucune autre demande ne subsiste, les fonctions adaptatives sont rétablies.

FIG. 7.1 - *Algorithme n° 6 : Communication directe entre deux esclaves sous MARS*

Les primitives suivantes pourront y pourvoir :

worker_communication_request : envoi au maître d'une demande d'entrée en section critique gelant temporairement les fonctions adaptatives de MARS pour permettre une communication directe entre esclaves (point n° 1 de l'*algorithme 6*). Le demandeur attend que le maître ait réalisé les points n^{os} 2 et 3 de cet algorithme, et reçoit le tableau des identificateurs PVM (« tids ») lui permettant d'adresser ses messages. Ce tableau est bien sûr un paramètre de sortie de la primitive.

worker_end_of_communication : envoi au maître de l'indication que l'application peut sortir de la section critique (point n° 5 de l'*algorithme 6*). Le maître exécute alors aussi le point n° 6.

Synchronisation des processus homogènes

Des primitives de synchronisation sont aussi indispensables. Elles permettront que l'attente d'une ressource par un ou plusieurs processus soit gérée proprement, en évitant en particulier l'attente active. Elle permettront aussi de gérer de façon simple les synchronisations globales entre tous les esclaves engagés.

Vus des processus MARS maître ou esclaves, ce sont les LRPC synchrones qui jouent ce rôle. Mais le problème n'est pas résolu pour autant car il se retrouve dans l'écriture de la partie « service » de ces LRPC. Pour l'écriture de cette dernière la primitives suivante peut être utile, afin d'alléger le code en encapsulant l'utilisation des sémaphores :

service_wait(condition) arrête l'exécution de la partie « service » d'un LRPC en attendant qu'une condition soit réalisée. Étant donné son caractère bloquant, cette

Souches :

```
PACK_REQ_STUB(LRPC_VECTOR)
    pvm_pklong(&arg->begin_sv, 1, 1);
    pvm_pklong(&arg->size_sv, 1, 1);
    pvm_pkdouble(arg->subvector,
                 arg->size_sv, 1);
END_STUB

UNPACK_REQ_STUB(LRPC_VECTOR)
    pvm_upklong(&arg->begin_sv, 1, 1);
    pvm_upklong(&arg->size_sv, 1, 1);
    pvm_upkdouble(arg->subvector,
                  arg->size_sv, 1);
END_STUB

PACK_RES_STUB(LRPC_VECTOR)
END_STUB

UNPACK_RES_STUB(LRPC_VECTOR)
END_STUB
```

Esclave :

```
static pthread_sem_t sem;
double size_vector; /* taille connue du vecteur */
double *vector;     /* vecteur à mettre à jour */

LRPC_REQ(LRPC_VECTOR) VECTOR_req;

LRPC_SERVICE(LRPC_VECTOR)
    int i;

    pthread_sem_P(&sem);
    for (i=0; i<req.size_sv; i++) {
        vector[req.begin_sv + i] = req.subvector[i];
    }
    pthread_sem_V(&sem);

    service_synchro(req.size_sv);
    /* libérer l'esclave quand tous les */
    /* morceaux du vecteur sont arrivés */
END_SERVICE(LRPC_VECTOR)
```

```
void worker_build_vector (double *subvector,
                          long begin_subvector,
                          long size_subvector)
/*
    effets de bord : le tableau « vector »
    est modifié par LRPC_VECTOR
*/

{
    int nb_slaves; /* nombre des esclaves */
    int *tids;     /* leurs identificateurs PVM */
    int i;


    worker_communication_request(&nb_slaves,
                                 &tids);
    /* points n°s 1, 2 et 3          */
    /* de l'algorithme 6 page 119 */


    /* phase de communication */
    VECTOR_req.begin_sv = begin_subvector;
    VECTOR_req.size_sv = size_subvector;
    VECTOR_req.subvector = subvector;

    for (i=0; i<nb_slaves - 1; i++) {
        ASYNC_LRPC(tids[i],
                   LRPC_VECTOR,
                   STD_PRIO,
                   MY_STACK_SIZE,
                   &VECTOR_req);
    }

    worker_wait(size_vector);
    /* attendre que le vecteur soit complet */

    /* points n°s 5 et 6 */
    /* de l'algorithme 6 */
    worker_end_of_communication();
}
```

FIG. 7.2 - *Implantation en C sous MARS de l'algorithme permettant un échange global des parties d'un vecteur distribué, sans passer par le maître*

primitive est à utiliser avec prudence, et il faut notamment veiller à ce que la condition puisse se réaliser au bout d'un temps raisonnable. Outre son utilisation pour faciliter la synchronisation avec un module externe d'une application hétérogène,

cette primitive pourrait aussi servir de base à l'implantation d'une synchronisation globale d'un module homogène, si les données transitent par le maître et à condition de tenir compte des remarques des paragraphes qui suivent. Par contre, elle ne suffira pas pour obtenir une synchronisation globale avec communication directe entre les esclaves.

Obtenir une synchronisation globale est un exercice délicat. C'est pourtant une opération qui se retrouve notamment dans les algorithmes numériques itératifs, où le passage à l'itération suivante exige la connaissance des résultats partiels calculés par tous les processus frères.

Mais un gros problème peut se poser si le dépliage de l'application n'est pas total, par exemple à cause d'un nombre de processeurs disponibles insuffisant pour la granularité choisie : dans ce cas, une partie des données n'a pas été distribuée par le maître et la synchronisation globale ne peut alors se traduire que par un « dead lock », car la condition de globalité sur les données ne pourra jamais être satisfaite. Pour y remédier, le blocage des threads esclaves sur le point de synchronisation doit rendre temporairement libre les machines qui les hébergent afin de permettre le dépliage de l'application concernant les données non encore traitées.

À l'heure actuelle, le système MARS ne permet pas le dépliage de l'application sur une station déjà occupée par un esclave MARS, même non actif car en attente par exemple sur un sémaphore ou un LRPC synchrone. Tant que le nombre de stations disponibles est suffisant ou que les applications ne présentent pas de point de synchronisation globale, le problème n'apparaît pas. Par contre rendre ce dépliage possible semble indispensable pour certaines applications, en particulier celles qui utilise des algorithmes numériques itératifs traitant d'importants volumes de données. Dans la suite, nous considérerons que cette modification a été apportée au système MARS, ce qui ne semble pas occasionner de difficulté majeure.

Évidemment, le programmeur d'une application présentant des synchronisations globales devra veiller à ce qu'aucune d'entre elles ne s'effectue dans une phase de communication qui inhibe le dépliage de l'application par le système MARS, (entre les appels aux primitives **worker_communication_request** et **worker_end_of_communication**) sinon le verrouillage serait total. Il est donc indispensable que ces synchronisations des esclaves ne s'effectuent qu'en dehors de ces sections critiques.

Deux cas sont à distinguer :

- les données transitent par le maître : la synchronisation est alors centralisée, et chaque esclave n'émet qu'un seul appel à un LRPC synchrone. La partie « service » de ce LRPC va assurer la redistribution des données, et la synchronisation par un appel à **service_wait**. Il n'y a pas de risque de blocage total dans ce cas car le maître ne pouvant pas être migré, il est inutile d'inhiber le dépliage de l'application.

- le LRPC est servi par un autre esclave : la synchronisation est distribuée. Elle sera assurée par les primitives suivantes :

  worker_wait(*ref*) bloque l'exécution du thread appelant tant que la valeur de contrôle est inférieure à *ref* (référence permettant de vérifier la condition de globalité). Cette valeur de contrôle (*val*) aura été établie au cours de l'appel au LRPC asynchrone générant l'échange de données sur lequel s'effectue la synchronisation. Par exemple, *val* pourra représenter le nombre de données traitées par l'esclave appelant le LRPC, et *ref* désigner le nombre total de données à traiter,

  service_synchro(*val*) est une primitive destinée à la partie service du LRPC appelé par les esclaves avant la primitive **worker_wait**. Elle ajoute *val* à la variable d'accumulation et contrôle que celle-ci reste inférieure à *ref*. Elle débloque l'esclave en attente dans le cas contraire.

Le protocole de communication et de synchronisation global sans transit des données par le maître, sera donc le suivant :

- appel de la primitive **worker_communication_request**,
- appel au LRPC d'envoi des données. Plusieurs appels successifs devront être faits (autant que d'esclaves frères).
  Il est évident que si chacun de ces appels est synchrone, le blocage sera total (« dead-lock »). Ces appels successifs devront donc obligatoirement être asynchrones (ASYNC_LRPC). Ceci implique que les données ne devront être transmises que comme paramètres d'entrée de ces LRPC, l'utilisation d'appels asynchrones ne rendant pas possible les paramètres de sortie. Dans le cas présent, concernant une synchronisation globale et des échanges du type « all-to-all », cette remarque n'est en aucune manière une restriction.
  Les souches de ce LRPC devront contenir, en plus des données à transférer, la valeur de contrôle *val* servant à déterminer l'achèvement de l'échange,
- appel de la primitive **worker_end_of_communication**.
- appel de la primitive **worker_wait**(*ref*), obligatoirement après la précédente pour éviter les blocages,
- à la fin de la partie « service » du LRPC d'échange des données, appel de la primitive **service_synchro**(*val*).

Un exemple d'utilisation de ces primitives de communication et de synchronisation, et en particulier d'un échange global de données sans transit par le maître, est la procédure SPMD décrite par la *figure 7.2.* Cette procédure effectue la reconstruction d'un vecteur dont le calcul a été réalisé de manière distribuée, opération qui revient souvent dans les méthodes numériques itératives exposées dans la *section 2.2.*

worker_build_vector : envoie à tous les processus frères (dont le le morceau de vecteur calculé localement, ainsi que sa taille et son emplacement dans le vecteur global.

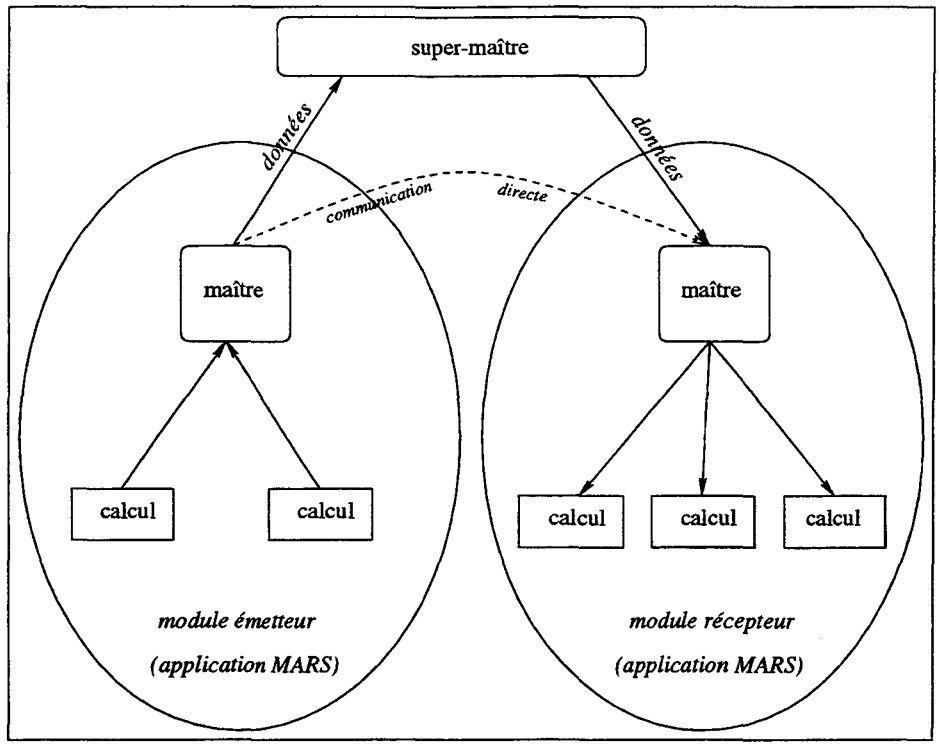


FIG. 7.3 - *Communications entre modules hétérogènes*

Celui-ci va se trouvé construit petit à petit par la partie « service » du LRPC appelé par tous les processus frères. La primitive **worker_build_vector** aura donc comme paramètre d'entrée le morceau de vecteur calculé localement, le vecteur entier étant une variable globale, accessible à la partie « service » du LRPC. Chaque processus aura donc une copie de ce vecteur pour la suite des calculs.

Communications entre modules hétérogènes

Les communications entre modules sont des transferts de données entre des applications MARS distinctes, communicant entre elles, et connues par le super-maître. Comme ces applications peuvent se dérouler sur des architectures très différentes, les données peuvent être représentées différemment, et devoir subir une conversion.

Comme cette hétérogénéité entre les modules alourdit les communications, celles-ci devront donc de toute façon être minimisées. Cela rend dans ce cas peu pénalisant le transit des données par le super-maître. De plus, l'intermédiaire du super maître se justifie par sa connaissance de la localisation et de l'état de tous les modules. Cependant, les données transférées proviennent en général d'un calcul effectué par les esclaves de l'un des modules, et elles seront finalement utilisées par les esclaves d'un autre module, dans un autre calcul. Le schéma global de ce transfert est donc conforme à la *figure 7.3.*

Sauf s'il tombe lui-même en panne, le maître d'une application MARS est parfaitement localisé, et n'est jamais migré. Il est donc possible d'éviter le transit des données par le super-maître *(communication directe en pointillés sur la figure 7.3)*, si celui-ci envoie aux maîtres des modules les informations nécessaires (« tids » PVM des maîtres des autres modules).

Les communications entre modules pourront donc se faire grâce aux primitives suivantes :

- pour connaître les autres modules :

  get_others_masters_tids sera une requête envoyée au super-maître par le maître de chaque module lors de l'initialisation. L'appelant recevra en retour le tableau des « tids » des maîtres de tous les modules engagés dans l'application hétérogène.

- pour transférer les données :
  Il sera fait appel à des LRPC sous PM^2 *(voir la section 7.2.3)*, les différents modules MARS utilisant déjà cette méthode pour leurs communications internes. La description des données à transférer est à la charge du programmeur dans les souches des LRPC.

Une attention particulière doit être portée aux problèmes de synchronisation lors de la réception d'un message inter-modules, particulièrement lorsqu'il s'agit d'une réception asynchrone. En effet, lors d'un traitement itératif, il importe que la prise en compte éventuelle de données externes au module soit faite à la même itération pour tous les esclaves servant le même algorithmes sur une structure de données répartie. Le non-respect de cette coordination rend le traitement incohérent, et peut même conduire à des « dead-locks ».

Une synchronisation est donc nécessaire au moment de la diffusion de ces données externes par le maître à tous ses esclaves. Mais en raison de l'asynchronisme des modules les uns par rapport aux autres, cette synchronisation n'intervient que si des données externes sont arrivées, au moment où les esclaves en ont besoin. Un protocole de synchronisation dans ce cas est donné par l'*algorithme 7 (figure 7.4)*. Ce protocole peut faire l'objet d'une primitive :

LRPC_get_external_data sera un LRPC synchrone appelé par chaque esclave pour interroger le maître sur l'arrivée de données issues d'un autre module. Ce LRPC bloquera l'esclave s'il est en avance sur les autres *(fin du point n° 2 de l'algorithme 7)*, et renverra un booléen indiquant si des données sont prêtes à être envoyées ou non. Le processus esclave appelant devra tester ce booléen, et appeler en cas de succès le LRPC de transfert de ces données, dont les souches *(voir la section 7.2.3)* dépendent évidemment de l'application.

ALGORITHME 7 : SYNCHRONISATION SUR UNE RÉCEPTION ASYNCHRONE

1. L'esclave n° i fait une requête pour demander si des données sont arrivées. Il précise dans sa requête son numéro d'itération courante nc. Cette requête est bloquante.
2. Si le maître n'a recu aucune donnée :

   Le maître met à jour son tableau de numéros d'itérations :
   $num\_it(i) = nc$;
   il indique à l'esclave qu'aucune donnée n'est arrivée et qu'il peut continuer son travail;

   sinon :

   Si le bloc de données disponible n'a été envoyé à aucun esclave ($send\_data\_to\_slave(i) == FALSE, \forall i$), alors :

   Calcul du numéro d'itération de la prise en compte :
   $num\_it\_data = 1 + \max(num\_it(i), \forall i)$;

   fin si;
   $num\_it(i) = nc$;
   Si $num\_it(i) < num\_it\_data$, alors :

   L'esclave peut continuer son travail : il n'a pas encore atteint l'itération de la prise en compte;
   Les données ne lui sont pas envoyées;

   sinon :

   Si $num\_it(i) == num\_it\_data$, alors :

   $send\_data\_to\_slave(i) = TRUE$;
   Les données sont envoyées à l'esclave n° i qui reprend son travail en en tenant compte;

   sinon :

   L'esclave a déjà pris ces données en compte, il attend que les autres en aient fait autant;
   $waiting\_slave(i) = TRUE$;

   fin si;

   fin si;

   fin si;
3. Si le bloc de données reçu a été envoyé à tous les esclaves ($send\_data\_to\_slave(i) == TRUE, \forall i$), alors (on se prépare à recevoir le bloc suivant) :

   $send\_data\_to\_slave(i) = FALSE, \forall i$;
   Les esclaves en attente (tels que $waiting\_slave(i) == TRUE$) sont libérés;
   $waiting\_slave(i) = FALSE, \forall i$;

   fin si;

FIG. 7.4 - *Algorithme n° 7 : Protocole de synchronisation des esclaves d'un module lors de la réception asynchrone de données issues d'un autre module*

Pour assurer la synchronisation des esclaves sur la réception de ces données disponibles de manière asynchrone et imprévisible, l'accès à ces données ne sera autorisé

par le maître que si :

- elles sont disponibles, c'est à dire déjà reçues par le maître.
- le numéro de l'itération courante de l'esclave appelant est égal au numéro de l'itération de prise en compte, celui-ci étant le suivant du plus grand jamais atteint par tous les esclaves avant l'arrivée des données.

Le blocage des esclaves ayant dépassé l'itération de prise en compte n'est en principe utile que si aucune autre synchronisation n'intervient entre eux au cours d'une itération (ce n'est pas le cas des applications numériques décrites dans les chapitres précédents).

Estampillage

Certaines communications, notamment asynchrones, ont besoin d'un estampillage, aussi bien entre modules hétérogènes qu'entre processus homogènes. Lorsque les nœuds de calcul sont chargés différemment, l'ordre d'arrivée des messages est indéterminé et des confusions risquent de se produire.

Ce problème peut être facilement résolu en ajoutant à toutes les primitives de communications un paramètre d'estampillage E_s (recevant par exemple le numéro de l'itération courante pour une méthode itérative), pouvant être remplacé par une valeur joker (par exemple 0) lorsque l'estampillage n'est pas nécessaire. L'identificateur PVM du message (« TAG ») pourra alors être calculé par le système en fonction de E_s et de l'identificateur fourni par le programmeur.

7.4 Vers un langage de contrôle

Le système MARS augmenté des primitives proposées dans la section précédente constitue un outil intéressant pour l'élaboration des programmes parallèles hétérogènes, mais au prix de certaines contraintes et de certaines lourdeurs. Il serait possible de les éviter et de simplifier la tâche du programmeur en construisant au-dessus de MARS un véritable langage de contrôle. Celui-ci profiterait des propriétés d'adaptativité du système précité tout en offrant des primitives de contrôle simples et claires, gommant au maximum les difficultés de mise en œuvre.

7.4.1 Encapsulation des fonctions de calcul

Lorsqu'on écrit une application d'une certaine complexité, il est important de pouvoir y intégrer des éléments ayant déjà été écrits, mis au point et testés par d'autres, afin de ne pas à chaque fois tout réécrire et, comme on dit, « réinventer la roue ». Les applications numériques font ainsi un grand usage de fonctions de calcul, en général séquentielles, déjà

développées et optimisées. Ces fonctions sont pour la plupart d'entre elles écrites en Fortran.

Or MARS et PM^2 sont écrits en C et il paraît difficilement concevable d'utiliser des environnements aussi élaborés autrement que par des programmes écrits dans le langage pour lequel ces environnements sont conçus, d'autant plus que la structure elle-même du programme source y est particulière : la construction des LRPC par exemple, ou la gestion de l'adaptativité exigent des protocoles bien précis [36, 28].

Pour réutiliser des fonctions de calcul existantes écrites dans un autre langage, il faudra donc encapsuler ces fonctions afin qu'elles soient « linkables » à un programme appelant en C. L'intégration de fonctions de calcul séquentielles est ainsi facilement réalisable.

Par contre, l'intégration de fonctions parallèles « toutes faites » n'est pas possible car elle mettrait en concurrence la parallélisation adaptative de MARS avec une autre façon de gérer le parallélisme, ce qui ne pourrait créer que des conflits. Le portage d'une application parallèle nécessite donc de repenser la parallélisation, mais le code séquentiel des éléments qui la composent peut être réutilisé.

Cela nécessite, en fait, de bien faire la distinction entre les fonctions de calculs, séquentielles sur chaque processeur (tout au moins en programmation MIMD), et les fonctions de contrôle chargées du lancement et de l'arrêt des tâches parallèles de calcul, de la distribution des données les concernant et des communications entre ces tâches. Cette distinction existe dans certains langages de contrôle tels que LC1 et LC2 [51] *(voir la section 3.2.4 page 43)* qui utilisent un langage spécifique pour le contrôle, et un langage classique pour le calcul. Cette solution, bien qu'extrême, a le mérite d'obliger le programmeur à faire une distinction nette entre contrôle et calcul, et par conséquent à écrire un programme plus clair. Elle a par contre l'inconvénient de nécessiter l'apprentissage d'une syntaxe particulière de A jusqu'à Z, au lieu d'une simple extension d'un langage classique, ce que font la plupart des langages de contrôle.

Un cas particulier est celui des tâches SIMD lancées sur une machine supportant ce type de parallélisme, au sein d'une application hétérogène parallèle. Ce type de machine est vue par MARS comme une entité unique, de la même manière qu'une station de travail séquentielle. Le code parallèle d'une fonction écrite pour une telle machine est donc intégrable au même titre qu'une fonction séquentielle, sous réserve des déclarations ad hoc destinées à l'éditeur de lien (sur la MasPar, par exemple, l'interfaçage entre le C et MP-Fortran nécessite des précautions particulières, l'adressage des données par le MPFortran étant spécifique). Évidemment, dans ce cas, tous les processeurs de la machine vont être engagés en même temps dans l'application, ou libérés en même temps si la machine se trouve chargée et que l'adaptativité joue. On a alors, avec ce type de machine, un comportement adaptatif à gros grain. Une gestion plus fine serait possible en partitionnant la machine en plusieurs sous-ensembles de processeurs élémentaires, ce que permettent d'ailleurs souvent les systèmes d'exploitation livrés par les constructeurs de ces machines. La prise en compte

de cette possibilité par MARS, qui devrait alors considérer chaque partition comme une sous-machine pseudo indépendante, exigerait des transformations importantes de ce système.

D'une manière générale, on a toujours intérêt, même si le langage utilisé par les tâches séquentielles élémentaires est aussi le C, à séparer, voire écrire dans un fichier à part les sources de telles tâches. Ici les tâches SIMD peuvent être traîtées de la même manière. Le code restant ne devrait plus être que du code de contrôle, contenant en particulier des appels à ces tâches, et aux primitives citées dans la *section 7.3.2.*

7.4.2 Problèmes liés à l'adaptativité

Un système adaptatif tel que MARS présente beaucoup d'intérêt car il permet de gérer au mieux la puissance de calcul disponible dans un parc de machines, et profiter des moments de sous-utilisation de certaines stations. Mais l'adaptativité ne fait pas bon ménage avec les algorithmes itératifs. La granularité des applications qui les utilisent est souvent assez limitée par la présence de nombreuses phases de communication, ce qui ne permet de jouer finement sur l'adaptativité. Mais surtout, les points de synchronisation (et en général il y en a au moins un par itération) posent un réel problème.

En effet, l'arrêt d'un esclave sur une station trop chargée aura pour effet de bloquer ses frères au prochain point de synchronisation tant que les données que cet esclave possédait n'auront pas été traitées. Cela implique l'attente qu'une station soit de nouveau disponible pour y créer un nouvel esclave, et cette attente peut être longue. De plus, il faudra que aussi attendre que les calculs entrepris par l'esclave tué soient refaits. Naturellement, l'algorithme peut être agencé pour que cette reprise soit minime, bien que ce type de programmation soit assez délicate.

Une solution grossière consiste à créer plus d'esclaves qu'il n'en faut, plusieurs restant en attente sur le LRPC de chargement des données. L'un d'entre eux sera débloqué par le maître si une station chargée nécessite de tuer l'esclave qu'elle hébergeait. Cette solution est moins coûteuse qu'il n'y paraît car les esclaves fonctionnant en « roue de secours » ne consomme pas de temps processeur durant leur attente, et ont un contexte très réduit tant que des données ne leur ont pas été distribuées. Cependant il n'est pas toujours possible d'avoir une réserve suffisante de stations libres. D'autre part, cela interdit l'installation d'autres processus MARS (venant d'autres applications, ou d'un autre module de la même application hétérogène) sur les stations concernées.

Une modification minimum du système MARS paraît s'imposer, pour autoriser au moins que lorsqu'un esclave est en attente sur un point de synchronisation, la station qu'il occupe puisse héberger un esclave frère, actif celui-là, qui recevrait les données à traiter en souffrance *(voir dans la section 7.3.2 la partie concernant la synchronisation des processus homogènes, à partir de la page 119)*. Dans ce cas, il n'y aurait plus de risque

d'attente très longue, voire de blocage, sur les points de synchronisation.

Une solution complémentaire serait d'adjoindre au système MARS une option permettant de restreindre l'adaptativité, afin que le pliage ou le dépliage de l'application ne puisse intervenir que pendant les moments ou le programmeur de l'application l'autorise. Cela permettrait, en particulier dans le cas des applications itératives :

- de laisser, au démarrage, le système choisir les stations les moins chargées,
- d'inhiber l'adaptativité en cours d'itération, ce qui préviendrait, à l'exception des pannes, tout problème de synchronisation dû au retrait d'un esclave. De plus cela permettrait d'alléger le protocole de communication directe entre esclaves *(voir l'algorithme 6, figure 7.1)*, leur identification pouvant être connue une fois pour toutes au début de l'itération,
- d'autoriser périodiquement une relance de l'adaptativité (par exemple à la fin de chaque itération), tout en sachant que cette étape pourrait entraîner une redistribution partielle des données, ce qui serait pénalisant pour certaines applications qui en manipulent des volumes importants.

L'inhibition partielle de l'adaptativité du système risque cependant d'être gênante en cas de panne d'une des stations, provoquant cette fois un blocage complet des processus esclaves au point de synchronisation. Pour préserver la tolérance aux pannes, il faudrait que le système réactive son adaptativité lorsqu'il détecte une telle panne. Ce point est délicat car cette réactivation ne peut pas s'effectuer à n'importe quel moment (en particulier, pas pendant une phase de communication directe entre esclaves, sous peine d'interrompre cette communication par un dépliage intempestif de l'application, et de rendre celle-ci incohérente).

7.4.3 Simplification de l'écriture des phases de communications

L'utilisation des LRPC pour la gestion des communications offre un intérêt certain *(voir la section 7.2.3)*, mais présente une certaine lourdeur dans son écriture. Ainsi, l'ajoût d'une phase de communication par exemple de l'esclave vers le maître pour transférer des données, nécessite de modifier quatre fichiers :

- le fichier de définition des souches (.h) contenant la liste des LRPC, et pour chacun la liste de leurs paramètres d'entrée et de sortie,
- le fichier d'écriture des souches (.c) contenant les instructions d'empaquetage et de dépaquetage des données,
- le fichier source de l'esclave qui contiendra la séquence d'appel au LRPC,
- le fichier source du maître qui contiendra la déclaration du LRPC et sa partie service.

Or l'écriture des souches peut parfaitement se faire de manière automatique à partir du moment où les données à transmettre sont définies. Le programmeur gagnera un temps appréciable s'il ne lui reste qu'à définir ce qui est vraiment de son ressort : La liste des données à transmettre et le traitement distant à effectuer sur ces données (la partie « service » du LRPC). Et encore, un certain nombre de ces traitements se retrouvent dans un grand nombre d'applications et pourraient être écrits une fois pour toutes (par exemple, pour les applications numériques, la reconstruction, la norme ou la somme d'un vecteur distribué). Dans certains cas même, ce traitement est absent, notamment dans une phase de communication pure ou de synchronisation entre esclaves.

Il serait donc intéressant de disposer de primitives simples, ayant une syntaxe intuitive, permettant d'utiliser des LRPC sous-jacents tout en allégeant leur mise en œuvre.

7.4.4 Spécifications possibles

Une solution serait la définition d'un langage spécifiquement de contrôle, analysé par un préprocesseur qui construirait les fichiers nécessaires à une compilation sous MARS. On disposerait ainsi d'un outil de programmation parallèle hétérogène d'accès simplifié, avec des primitives adaptées aux opérations de contrôle les plus utilisées dans les applications, sans perdre aucun des avantages de MARS. Le programmeur désireux d'affiner son application pourrait toujours reprendre les fichiers générés et les travailler directement sous MARS. Il disposerait ainsi de possibilités étendues, mais au prix d'une complexité de programmation accrue.

Avec un tel langage, dissocié des contraintes d'implantation liées au système sous-jacent, il serait possible de réutiliser certains concepts intéressants présents dans des langages tels que LC1 et LC2 [51], qui généralisent la notion de divers niveaux de mémoire utilisée dans les machines à mémoire partagée. Cette abstraction pourrait ici encapsuler complètement les systèmes sous-jacents :

- la mémoire locale constitue le premier niveau, et est accessible directement par les fonctions de calcul, séquentielles sur chaque processeur,
- la mémoire distante utilisée par la même application homogène (par exemple au sein de la même machine parallèle ou dans des stations séquentielles en réseau impliquées dans le même module sous MARS) constitue le second niveau. Son accès se fait à travers des primitives du langage de contrôle qui seront implantées par des LRPC,
- la mémoire associée à un autre module, séquentiel ou parallèle, d'une application hétérogène constitue le troisième niveau. Son accès utilise des primitives de communications inter modules hétérogènes, implantées comme des communications entre les maîtres d'applications MARS distinctes,
- enfin, la mémoire de masse constitue un quatrième niveau.

Définition des souches du LRPC (fichier .h) :

```
LRPC_DECL_REQ(LRPC_GET_MEM2_x,
        )
LRPC_DECL_RES(LRPC_GET_MEM2_x,
        double x;)
```

Appel du LRPC dans l'esclave demandeur de la donnée à transférer :

```
LRPC(tid, LRPC_GET_MEM2_x,
     STD_PRIO, MY_STACK_SIZE,
     NULL, &GET_MEM2_x_res);
x = res.x;
```

Implantation des souches du LRPC (fichier .c) :

```
PACK_REQ_STUB(LRPC_GET_MEM2_x)
END_STUB

UNPACK_REQ_STUB(LRPC_GET_MEM2_x)
END_STUB

PACK_RES_STUB(LRPC_GET_MEM2_x)
  pvm_pkdouble(&arg->x, 1, 1);
END_STUB

UNPACK_RES_STUB(LRPC_GET_MEM2_x)
  pvm_upkdouble(&arg->x, 1, 1);
END_STUB
```

Dans le fichier source de l'esclave fournisseur de la donnée réclamée :

```
/* Déclaration et initialisation du sémaphore d'exclusion mutuelle pour accéder à x */
  pthread_sem_t MUTEX_x;
  pthread_sem_init(MUTEX_x, 0);

/* Partie service du LRPC */
  LRPC_SERVICE(LRPC_GET_MEM2_x)
    pthread_sem_P(&MUTEX_x);
    res.x = x;
    pthread_sem_V(&MUTEX_x);
  END_SERVICE(LRPC_GET_MEM2_x)
```

FIG. 7.5 - *Implantation sous MARS de la primitive « get_mem2(tid,'double',x) » assurant la copie de la variable x, de type réel double précision, depuis la mémoire de second niveau vers la mémoire locale (donc de premier niveau).*

La *figure 7.5* montre l'implantation, avec le détail dans les différents fichiers des écritures exigées par le système, de la primitive **get_mem2** assurant la recopie d'une donnée de la mémoire vue de niveau 2 par un esclave vers sa mémoire de niveau 1. L'appel de cette primitive présuppose un appel précédent à **worker_communication_request**, et d'avoir sélectionné l'esclave à appeler. Naturellement, après obtention de la copie, un appel à **worker_end_of_communication** est nécessaire.

Un niveau d'abstraction plus important consisterait à affecter à toutes les données une adresse globale, comme le fait Nexus [21] (voir la section 3.2.3). Mais l'adaptativité de MARS rend la localisation des données distribuées dynamique. L'accès à une donnée distante nécessiterait alors un accès préalable au maître (dans le cas d'une donnée de niveau 2) ou au super-maître (pour une donnée de niveau 3) afin de localiser l'esclave qui la détient. De plus, l'adaptativité devra être temporairement inhibée pour que cette localisation reste valide jusqu'à la fin de l'opération d'accès.

Définition des souches du LRPC (fichier .h):

```
LRPC_DECL_REQ(LRPC_SYNC_a_b,
        int val;)
LRPC_DECL_RES(LRPC_SYNC_a_b,
        )
```

Implantation des souches du LRPC (fichier .c):

```
PACK_REQ_STUB(LRPC_SYNC_a_b)
  pvm_pkint(&arg->val, 1, 1);
END_STUB

UNPACK_REQ_STUB(LRPC_SYNC_a_b)
  pvm_upkint(&arg->val, 1, 1);
END_STUB

PACK_RES_STUB(LRPC_SYNC_a_b)
END_STUB

UNPACK_RES_STUB(LRPC_SYNC_a_b)
END_STUB
```

Appel du LRPC dans le programme où se trouve l'étiquette **a**:

```
req.val = 1;
LRPC(Parent_tid, LRPC_SYNC_a_b,
     STD_PRIO, MY_STACK_SIZE,
     &SYNC_a_b_req, NULL);
```

Appel du LRPC dans le programme où se trouve l'étiquette **b**:

```
req.val = 0;
LRPC(Parent_tid, LRPC_SYNC_a_b,
     STD_PRIO, MY_STACK_SIZE,
     &SYNC_a_b_req, NULL);
```

Dans le fichier source du maître:

```
/* Déclaration et initialisation du sémaphore */
pthread_sem_t SYNC_a_b;
pthread_sem_init(SYNC_a_b, 0);

/* Partie service du LRPC */
LRPC_SERVICE(LRPC_SYNC_a_b)
  if (req.val) pthread_sem_V(&SYNC_a_b);
  else pthread_sem_P(&SYNC_a_b);
END_SERVICE(LRPC_SYNC_a_b)
```

FIG. 7.6 - *Implantation sous MARS de la primitive « ensure_order(a,b) » permettant le respect de la précédence entre les étiquettes a et b de processus homogènes.*

Un autre concept indispensable pour un langage de contrôle, est celui de la programmation du graphe de précédence. Le cas de l'exclusion mutuelle est prévu par PM^2 qui possède les primitives **pm2_enter_critical_section()** et **pm2_leave_critical_section()**. D'autre part une synchronisation sur des étiquettes, telle que celle développée en LC2 *(voir la figure 3.3 et le descriptif de la page 46)*, peut facilement être implantée sous MARS, mais son écriture par le programmeur de l'application nécessite encore une fois beaucoup de manipulations (utilisation d'un LRPC, décrit par quatre fichiers, comme indiqué dans la section 7.4.3), et utilisation de sémaphores. Ce travail peut être très allégé dans le cas d'un langage de contrôle permettant l'écriture automatique des fichiers MARS à partir de primitives spécifiques. La *figure 7.6* montre l'implantation, dans les différents fichiers

MARS, de la primitive **ensure_order(a, b)** qui permet d'obtenir, pour deux processus d'un même module MARS, que l'exécution de ce qui suit l'étiquette **b** attende la fin de l'exécution de ce qui précède l'étiquette **a**. Dans le cas de modules hétérogènes, les étiquettes devront concerner les maîtres des deux modules à synchroniser. La seule différence dans l'implantation est que la partie « service » du LRPC appelé sera exécutée par le super-maître, et que l'identificateur « Parent_tid » des appels à ce LRPC sera alors l'identificateur du super-maître.

7.5 Expérimentation

Pour évaluer la pertinence des spécifications précitées, il était nécessaire d'implanter, sous MARS augmenté des primitives proposées, des applications déjà écrites sous un autre système, afin de pouvoir faire des comparaisons. Notre point de départ étant les méthodes numériques itératives, la méthode GMRES(m), méthode éprouvée s'il en est, et moins compliquée à mettre en œuvre que la méthode hybride décrite aux *chapitres 4 et 5*, a semblé être un bon choix.

Deux versions ont été réalisées :

- la première (dite « MARS-centralisée ») est une transcription, en langage C sous MARS, de l'algorithme GMRES(m) selon la stricte philosophie maître-esclaves du système. Ainsi les communications entre esclaves transitent par le maître par des LRPC synchrones, le maître assurant la cohérence des données transmises et la synchronisation de l'ensemble des esclaves.
- la seconde (dite « MARS-distribuée ») utilise, en plus du système MARS, les primitives de communication et de synchronisation des processus homogènes décrits dans la *section 7.3.2*. Ainsi les communications entre esclaves ne transitent pas par le maître, et la synchronisation entre esclaves est elle-même distribuée. Le maître n'intervient que relativement à l'adaptativité du système, ainsi que comme serveur d'adresses avant toute nouvelle phase de communications.

Il est évident que le surcoût du système MARS apparaît essentiellement dans les phases de communications, et ceci pour les deux versions. Il dépend surtout du rapport entre le volume des communications et celui des calculs, et il se fera d'autant plus sentir que ce rapport sera faible. Pour se placer dans un cas intéressant, une matrice de grande taille a été choisie (matrice « vp », $n = 10000$), avec une taille de sous-espace moyenne ($m = 40$) déterminant un volume de communications non négligeable mais pas excessif.

Les deux versions sous MARS ont été testées dans les mêmes condition que la version Fortran-PVM servant de référence : la même matrice avec les mêmes paramètres, le même type de parallélisme (SPMD), la même machine (la ferme d'Alphas), le même nombre de processeurs ($p = 4$).

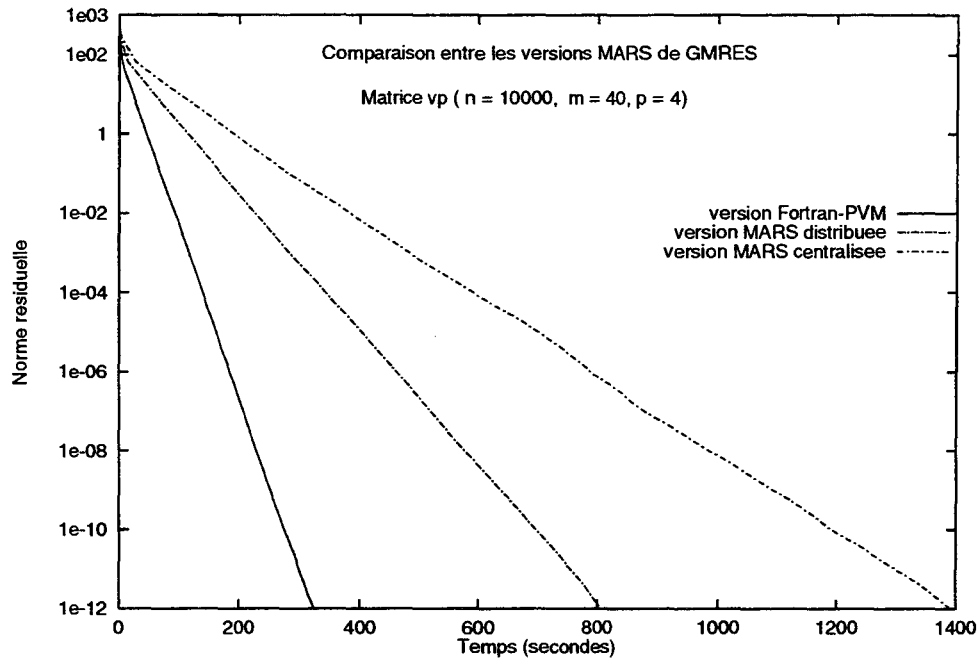


FIG. 7.7 - *Évolution de la norme résiduelle avec la méthode GMRES(m) pure sous MARS (version centralisée et version distribuée) comparée avec la version Fortran-PVM*

Les résultats apparaissant sur la *figure 7.7* semblent décevants car ils traduisent un surcoût prohibitif du système. Ils appellent cependant les remarques suivantes :

- la version « MARS-centralisée » est la plus pénalisante et ce n'est pas une surprise. Le gain de temps obtenu par les communications directes entre esclaves dans la version « MARS-distribuée » justifie la complexification du code engendrée par cette version. Cette difficulté est d'ailleurs transparente à l'utilisateur des primitives de contrôle.

- les écarts entre les différentes versions risquent d'être très dépendants de la matrice utilisée, en particulier en fonction de son nombre d'éléments non nuls et de la taille du sous-espace de Krylov choisie.

- les temps importants de développement des différentes composantes de MARS ont fait que la version actuelle s'appuie sur une version de PM^2 qui n'est pas la plus récente. Une version future, s'appuyant sur des protocoles de communication plus performants, devrait permettre d'obtenir de meilleurs résultats.

- il est certain que les méthodes itératives sont très éloignées du type d'application pour lequel MARS a été conçu car elles contiennent trop de dépendances globales. De plus, le principe de la répartition du travail par consommation des données n'est pas applicable lorqu'on itère sur ces mêmes données.

- l'intérêt de MARS est surtout l'adaptativité. Même si le surcoût est important, l'utilisation des périodes d'inactivité des stations de travail d'un réseau peut compenser très largement cet inconvénient.

7.6 Conclusion

Il a paru intéressant, compte-tenu de la capacité de l'environnement MARS de gérer des applications parallèles hétérogènes, et de ses propriétés d'adaptativité, de baser un langage de contrôle sur cet environnement. Les primitives déjà présentes sous MARS et sous l'environnement multithreadé PM^2 sur lequel il est construit permettent déjà une gestion efficace des applications hétérogènes.

Cependant la nature centralisée du système maître-esclave sous-jacent (un seul maître et plusieurs esclaves identiques) ne répond qu'imparfaitement aux besoins d'applications numériques parallèles hétérogènes telles que celles décrites dans les chapitres précédents, notamment à cause du volume des échanges de données entre les processus de calcul et de la diversité des fonctions attribuées aux processus affectés sur des machines d'architectures différentes. Le goulet d'étranglement causé par l'obligation du transit des données par le maître doit particulièrement pouvoir être évité.

Mais il suffit d'un petit nombre de primitives supplémentaires pour résoudre ces inconvénients, et étendre les possibilités offertes par cet environnement pour construire des applications parallèles hétérogènes dont la structure soit conforme au modèle décrit au *chapitre 6*. Naturellement, la liste des primitives proposées n'est pas exhaustive. Elle peut notamment être enrichie de tout protocole se retrouvant souvent dans les applications hétérogènes parallèles, numériques ou non.

Quelques modifications, liées au degré d'adaptativité du système, sont proposées. Elles seront surtout utiles pour éviter des blocages dans le cas d'applications où les synchronisations globales sont fréquentes, et notamment pour les applications itératives.

Pour faciliter le travail du programmeur, réutiliser les bibliothèques de calcul séquentiel existantes et automatiser l'écriture relativement lourde des fichiers sources en C sous MARS, il est possible d'élaborer un langage de contrôle qui intègre le modèle de parallélisme hétérogène précédemment décrit et les primitives s'y rapportant. Le système MARS sous-jacent permet d'y adjoindre facilement des abstractions concernant l'accès aux données distantes ou le respect d'un graphe de précédence.

Les applications implantées sous forme parallèle hétérogène le sont souvent en raison de leurs propriétés d'asynchronisme et de granularité à plusieurs niveaux. Leur réalisation par des modules MARS asynchrones permet de tirer le meilleur parti des ressources disponibles à un instant donné.

Une illustration intéressante en est la méthode parallèle asynchrone GMRES(m)/LS-Arnoldi décrite au *chapitre 4*. Nous avons vu qu'en raison de la limitation du degré de parallélisme de la méthode GMRES(m) à cause de l'importance du volume des communications, il était possible d'accélérer la convergence en construisant une méthode hybride, dont les modules étrangers à l'algorithme GMRES(m) seraient distribués sur d'autres nœuds de calcul, augmentant par là même la parallélisation. Nous avions alors remarqué que ces modules nécessitait une puissance de calcul moindre que celle affectée à l'algorithme principal. Cependant leur coût en terme de temps de calcul n'est pas négligeable, même si l'accélération qui en résulte le justifie. Or, grâce aux propriétés de MARS qui permettent notamment de travailler sur un réseau de stations séquentielles « à l'insu » de leur utilisateur principal, en profitant des temps d'inutilisation de la console, on pourrait concevoir de distribuer ces modules sur un tel réseau, afin de profiter ainsi d'une augmentation du degré de parallélisme conduisant à une accélération de la convergence quasi gratuite.

Les applications de ce type peuvent donc être particulièrement mises en valeur par l'adaptativité de MARS, pourvu que le programmeur dispose d'outils suffisamment puissants et adaptés pour une mise en œuvre simple.

Chapitre 8

Conclusion et Perspectives

Dans un grand nombre de problèmes scientifiques d'envergure utilisant des structures de données de grande taille, la majeure partie du temps de calcul est consacrée à la résolution de systèmes linéaires. C'est pourquoi il est important d'accorder à l'amélioration de ce type de résolution l'importance qu'elle mérite.

La première partie de cette thèse s'est attachée à mettre en œuvre des méthodes numériques itératives, pour des problèmes décrits par des matrices creuses de grande taille (tenant cependant en mémoire vive), dans des environnements parallèles hétérogènes. Après avoir présenté l'intérêt des méthodes itératives par rapport aux méthodes directes, notamment à propos des matrices creuses, la méthode GMRES(m) a été choisie comme base de ce travail. Celui-ci a été réalisé en collaboration avec Azeddine Essai, du laboratoire d'Analyse Numérique et d'Optimisation (ANO), dont la thèse, « Méthode hybride parallèle hétérogène et méthodes pondérées pour la résolution des systèmes linéaires » [13], développe les aspects numériques de la question.

Ayant d'abord réalisé une version parallèle de la méthode GMRES(m) pure qui nous a servi de référence, et ayant remarqué ses limitations quant à la granularité du parallélisme, nous avons développé ensuite la méthode hybride GMRES(m)/LS-Arnoldi, dans laquelle les valeurs propres obtenues par la méthode d'Arnoldi sont utilisées par la méthode « Least Squares » pour obtenir un itéré permettant un meilleur redémarrage de la méthode GMRES(m), et donc une accélération de la convergence.

L'implantation parallèle de cette méthode hybride a été réalisée selon deux modèles, sur deux sites offrant les accès à des machines parallèles très différentes. Une version, utilisant deux machines parallèles (une ferme d'Alphas et une MasPar MP1) et deux stations séquentielles, exploite au maximum l'hétérogénéité du réseau et l'asynchronisme des processus. Elle a de ce fait été qualifiée de « parallèle hétérogène asynchrone ». L'autre, n'utilisant qu'une machine parallèle (une CM5) et trois stations de travail séquentielles, fait alterner sur la même machine les parties parallèles des différentes méthodes numériques employées, et a été qualifiée de « parallèle entrelacée ».

Au point de vue numérique, les deux versions sont construites sur la même méthode hybride, et ont un comportement similaire : on observe dans les deux cas une augmentation temporaire de la norme résiduelle lors d'une prise en compte de valeurs propres par la méthode « Least Squares », suivie d'une diminution significative engendrant une diminution du nombre d'itérations nécessaire à la convergence. Globalement donc, la résolution du système s'en trouve accélérée, bien que le nombre d'itérations gagnées dépendent énormément de la matrice utilisée et du choix judicieux des paramètres, en particulier de la taille de l'espace de projection. C'est encore plus vrai en termes de durée du calcul, la durée de la prise en compte « Least Squares » s'avérant souvent largement supérieure à celle d'une itération GMRES(m), ce surcoût étant surtout liés aux paramètres k (degré du polynôme Least Squares) et l (nombre d'itérations Least Squares à chaque prise en compte). Les seuls cas cependant où la méthode hybride se soit trouvée en défaut avec une durée de calcul supérieure à celle obtenue avec la méthode GMRES(m) pure, correspond au cas particulier où $l = 1$.

Il est difficile de comparer les temps de calcul obtenus par les deux versions réalisées, car les machines parallèles utilisées ont des performances trop différentes. Cependant il est évident que la version parallèle entrelacée est pénalisée par des attentes aux points de synchronisation induits par l'entrelacement des parties parallèles des algorithmes. La version parallèle hétérogène asynchrone, plus souple, autorise chacun des algorithmes numériques qui la composent à travailler au mieux des possibilités instantanées de sa machine hôte, en fonction de sa charge du moment. L'asynchronisme du couplage des différents composants logiciels induit un non déterminisme du déroulement du calcul sans influence sur le résultat numérique final (la solution du système linéaire), mais pouvant influer sur la rapidité de la convergence.

Pour chacune de ces versions, les performances dépendent d'un grand nombre de paramètres, dont les valeurs donnant les meilleurs résultats varient beaucoup selon la matrice considérée, et dont l'étude exhaustive prendrait un temps sans commune mesure avec les améliorations qu'on pourrait en espérer. L'influence des paramètres les plus importants, tels que les tailles des espaces de projection ou les paramètres k et l précités, a permis de mettre en évidence les conditions d'obtention de performances satisfaisantes. Mais celles-ci peuvent certainement être améliorée par une étude plus fine de l'ensemble de ces paramètres.

Ces deux implantations de la méthode hybride ne sont pas les seules possibles et plusieurs autres solutions ont été passées en revue, le but étant d'être capable, pour un ensemble de machines donné d'un réseau autorisant le calcul parallèle, d'élaborer rapidement, à partir de modules logiciels de base, une implantation optimale pour les architectures concernées.

La généralisation de ce principe aux autres applications parallèles hétérogènes a conduit

à s'intéresser aux moyens de concevoir et d'écrire de telles applications et donc aux outils le permettant, en particulier les langages de contrôle. Or s'il existe de nombreux environnements pour la mise en œuvre du parallélisme, rares sont ceux qui prennent en compte l'hétérogénéité. Encore dans ce cas le déterminisme est-il de rigueur, alors que le contraire peut se révéler intéressant, ce qui était le cas de l'application numérique étudiée précédemment. De plus, la plupart ne prennent pas en compte les variations de charge des machines, qui cependant influent énormément sur les performances des systèmes.

Ce dernier point a suscité des développements récents permettant la répartition de charge dynamique et la migration des processus en environnement distribué. En particulier le système MARS, basé sur l'environnement multithreadé PM^2, autorise la construction d'applications bénéficiant d'un parallélisme adaptatif, permettant une granularité et une distribution des données dynamiques en fonction de la charge des machines. Ses potentialités importantes en font une base intéressante pour la définition d'un langage de contrôle facilitant la mise en œuvre d'applications parallèles hétérogènes, éventuellement asynchrones. Les spécifications d'un certain nombre de primitives ont été décrites dans le dernier chapitre, afin d'étendre dans cette direction les possibilités de MARS.

La poursuite de ce travail offre des perspectives dans plusieurs domaines. D'autres méthodes numériques hybrides peuvent naturellement être élaborées et la méthode présentée, GMRES(m)/LS-Arnoldi, peut elle-même être améliorée. Si une bonne partie de ces recherches relève de l'analyse numérique, leur mise en œuvre dans des environnements parallèles, voire hétérogènes, est loin d'être triviale, et il y a encore beaucoup à faire dans ce domaine.

Le langage de contrôle basé sur MARS dont les spécifications ont été définies dans le *chapitre 7* doit être développé, en particulier le préprocesseur permettant la construction automatique des différents fichiers utilisés par la compilation sous MARS, à partir d'un texte source contenant des primitives de contrôle. Ce langage a aussi besoin d'être validé par l'élaboration d'applications parallèles hétérogènes judicieuses l'utilisant. Les performances obtenues devront être analysées, aussi bien sur le plan du temps de calcul qu'à propos de la granularité et de son contrôle, ainsi que de l'utilisation des ressources.

Ce langage de contrôle pourrait aussi être reformulé selon le modèle « orienté objets », ce qui permettrait une conception des programmes hétérogènes plus rigoureuse et plus cohérente. De plus, cette optique se rapprocherait de la tendance actuelle de l'évolution des langages de programmation. D'ailleurs, la décomposition en modules séquentiels ou parallèles, chacun ayant son code et ses données, ainsi que l'existence de plusieurs niveaux de programmation, sont tout à fait dans l'esprit de la programmation « orientée objets ».

Un travail de modélisation est aussi nécessaire afin de dégager les abstractions nécessaires, notamment quant au parallélisme hétérogène et au couplage des différents composants logiciels, afin de rendre la mise en œuvre de ce type de programme plus simple et

plus claire.

La généralisation des réseaux, de plus en plus rapides, interconnectant des machines parallèles et séquentielles, récentes et anciennes, et l'émergence de problèmes industriels exploitant des volumes de données considérables, amènent un nombre de plus en plus grand d'applications, numériques ou non, à être pensées en termes de parallélisme hétérogène. Il va sans dire que les efforts de développement et de modélisation consentis dans ce sens offrent des perspectives de recherche importantes.

Bibliographie

[1] W. E. Arnoldi. The principale of minimized iteration in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9:17–29, 1951.

[2] G. Bergère. Couplage d'algorithmes numériques hybrides parallèles. *Séminaires jeunes chercheurs de l'axe "Informatique Parallèle et Distribuée", Actes des journées 1996*, pages 29–36. LIFL, Université de Lille I, 1996. LIFL report AS-171.

[3] G. Bergère, A. Essai, and S. Petiton. Utilisation asynchrone de machines parallèles hétérogènes pour l'accélération de la résolution de systèmes linéaires. *Actes de RenPar'9*, Lausanne, 1997.

[4] G. Bergère, A. Essai, and S. Petiton. Asynchronous parallel hybrid gmres/ls-arnoldi method. Technical Report 388, Laboratoire d'Analyse Numérique et d'Optimisation, Université de Lille I, 1998.

[5] R. Boisvert. A web resource for test matrix collections. Townmeeting on Online Delivery of Nist Reference Data, NIST, Gaithersburg, May 30, 1997.

[6] F. Coelho. *Contributions à la compilation du High Performance Fortran.* Thèse de l'École des Mines de Paris, 1996.

[7] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles.* InterEditions, 1993.

[8] E. de Sturler and H.A. van der Vorst. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *Applied Numerical Mathematics*, 18(4):441, 1995.

[9] G. Edjlali. *Contribution à la parallélisation de méthodes itératives hybrides pour matrices creuses sur architectures hétérogènes.* Thèse de l'Université de Paris VI, 1994.

[10] G. Edjlali, S. Petiton, and N. Emad. Interleaved parallel hybrid arnoldi method for a parallel machine and a network of workstations, 1996.

[11] G. Edjlali, A. Sussman, and J. Saltz. Interoperability of data parallel runtime libraries. Department of Computer Science, University of Maryland, 1997.

[12] J. Erhel. A parallel GMRES version for general sparse matrices. *Electronic Transactions on Numerical Analysis*, 3:160–176, 1995.

[13] A. Essai. *Méthode hybride parallèle hétérogène et méthodes pondérées pour la résolution des systèmes linéaires.* Thèse de l'Université de Lille I, ANO, 1999.

[14] A. Essai, G. Bergère, and S. Petiton. Heterogeneous parallel hybrid gmres/ls-arnoldi method. *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[15] B. Folliot, P. Sens, and P.-G. Raverdy. Plate-forme de répartition de charge et de tolérance aux fautes pour applications parallèles en environnement réparti. *Calculateurs Parallèles*, 7(4):345–366, 1995.

[16] I. Foster. Task parallelism and high-performance languages. *IEEE Parallel and Distributed Technology*, 2(3):39–48, 1994.

[17] I. Foster and K. Mani Chandy. Fortran M language definition. Technical Report ANL-93/28, Mathematics and Computer Science Division, Argonne National Laboratory, 1993.

[18] I. Foster, C.Kesselman, and S. Tuecke. Nexus: Runtime support for task-parallel programming languages. Technical Memorandum 205, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[19] I. Foster, J. Garnett, and S. Tuecke. Nexus user's guide. Technical Memorandum 204, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[20] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high performance networked computing systems. *Journal of parallel and distributed computing*, 40:35–48, 1997.

[21] I. Foster and C. Kesselman. Language constructs and runtime systems for compositional parallel programming. Preprint MCS-P489-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[22] I. Foster, R. Olson, and S. Tuecke. Programming in Fortran M. Technical Report ANL-93/26, Mathematics and Computer Science Division, Argonne National Laboratory, 1993.

[23] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

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

[25] A. Greenbaum. Estimating the attainable accuracy of recursively computed residual methods. *SIAM J. Matrix Anal. and Appl.*, 18(3):535–551, 1997.

[26] Z. Hafidi. *MARS: Un environnement de programmation parallèle adaptative dans les réseaux de machines hétérogènes multi-utilisateurs.* Thèse de l'Université de Lille I, LIFL, 1998.

[27] Z. Hafidi, E.-G. Talbi, and J.-M. Geib. MARS: Un ordonnanceur adaptatif d'applications parallèles dans un environnement multi-utilisateurs. Laboratoire d'Informatique Fondamentale de Lille, Université de Lille I, 1996.

[28] Z. Hafidi, E.-G. Talbi, and J.-M. Geib. Méta-systèmes: Vers l'intégration des machines parallèles et les réseaux de stations hétérogènes. Laboratoire d'Informatique Fondamentale de Lille, Université de Lille I, 1998.

[29] High Performance Fortran forum. *High Performance Fortran Language Specification*, 1997.

[30] D. Kebbal, E.-G. Talbi, and J.-M. Geib. A new approach for checkpointing parallel applications. *Proceedings of the international conference on parallel and distributed processing technique and applications (PDPTA'97)*, 1997.

[31] D. Lazure. *Programmation géométrique à parallélisme de données: modèle, langage et compilation.* Thèse de l'Université de Lille I, LIFL, 1995.

[32] J. P. Malmquist and E. L. Robertson. On the complexity of partitioning sparse matrix representations. *BIT*, 24:60–68, 1984.

[33] P. Marquet. Langages et expression du parallélisme de données. *Technique et science informatique*, 12(6):685, 1993.

[34] R. S. Martin, G. Peters, and J. H. Wilkinson. The QR algorithm for real hessenberg matrices. *Numerical Mathematics*, 1970.

[35] R. Namyst and J.F. Mehaut. PM^2: Parallel Multithreaded Machine, a computing environment for distributed architectures. *Parco'95 proceedings*, Gent, 1995.

[36] R. Namyst and J.F. Mehaut. PM^2: Parallel Multithreaded Machine, guide d'utilisation, version 1. LIFL, Université de Lille I, 1995.

[37] A. T. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM J. Sci. Comput.*, 14(3):519–530, 1993.

[38] S. Petiton. *Du développement de Logiciels Numériques en Environnements Parallèles.* Thèse de l'Université de Paris VI, 1988.

[39] S. Petiton. Parallel QR algorithm for iterative subspace methods on the connection machine (CM2). *In J. Dongarra, P. Messina, D. Sorensen, and R. Voigt, editors, Parallel Processing for Scientific Computing*, SIAM, 1990.

[40] S. Petiton. Parallel subspace method for non-hermitian eigenproblems on the connection machine (CM2). *Applied Numerical Mathematics*, 10(1):19, 1992.

[41] S. Petiton. Data parallel sparse matrix computation on CM2 and CM5 for iterative methods. University of Minnesota, AHPCRC, 1993.

[42] S. Petiton. *Contribution à une méthodologie globale pour le calcul scientifique parallèle.* Thèse d'habilitation, Université de Paris VI, 1993.

[43] S. Petiton, G. Bergère, and G. Edjlali. Méthodes hybrides pour réseaux de machines parallèles hétérogènes. *Ecole Française de parallélisme, réseaux et systèmes, 1-5 juillet 1996, Presqu'île de Giens, France*, INRIA, 1996.

[44] M. Ranganathan, A. Acharya, G. Edjlali, A. Sussman, and J. Saltz. Runtime coupling of data-parallel programs. International Conference on Supercomputing, ICS'96, 1996.

[45] Y. Saad. Etude de la convergence du procédé d'Arnoldi pour le calcul d'éléments propres de grandes matrices non symétriques, 1979.

[46] Y. Saad. Least squares polynomials in the complex plane and their use for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:155–169, 1987.

[47] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations. RIACS, NASA Ames Research Center, 1991.

[48] Y. Saad. *Numerical methods for large eigenvalues problems.* Manchester University Press Series in Algorithms and Architectures for Advanced Scientific Computing, 1993.

[49] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[50] J. Saltz, S. Petiton, H. Berryman, and A. Rifkin. Performance effects of irregular communication patterns on massively parallel multiprocessors. *Journal of Parallel and Distributed Computing*, 13(2):202, 1991.

[51] SINTRA. *Manuel d'utilisation du langage LC2*, 1985.

[52] A. Sussman. Execution models for mapping programs onto distributed memory parallel computers. *ICASE*, 92-8, 1992.

[53] H. F. Walker. Implementation of the GMRES method using householder transformations. *SIAM J. Sci. Statist. Comput.*, 9:152, 1988.

[54] L. F. Wilson. Analysis of algorithmic structures with heterogeneous tasks. *ICASE*, 96-1, 1996.

Table des figures

Chapitre 2 : Résolution de systèmes linéaires non symétriques de très grande taille

Chapitre 3 : Outils pour la programmation parallèle hétérogène

Chapitre 4 : Algorithme parallèle hétérogène asynchrone

Chapitre 5 : Algorithme parallèle entrelacé

Chapitre 6 : Construction modulaire d'applications parallèles hétérogènes

Chapitre 7 : Expression du parallélisme hétérogène par un langage de contrôle